Communicating Process Architectures 2001
Alan Chalmers, Majid Mirmehdi and Henk Muller (Eds.)
IOS Press, 2001

Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment

F.R.M. Barnes and P.H. Welch
Computing Laboratory, University of Kent, Canterbury, KENT. CT2 7NF
{frmb2,phw}@ukc.ac.uk

Abstract. Traditional imperative languages (such as C) and modern object-oriented languages are plagued by uncontrolled resource aliasing problems. Add in concurrency and the problems compound exponentially. Improperly synchronised access to shared (i.e. aliased) resources leads to problems of race-hazard, deadlock, livelock and starvation.

This paper describes the binding into occam (a concurrent processing language based on CSP) of a secure, dynamic and efficient way of sharing data between parallel processes with minimal synchronisation overheads. The key new facilities provided are: a movement semantics for assignment and communication, strict zero-aliasing, apparently dynamic memory allocation and automatic zero-or-very-small-unit-time garbage collection. The implementation of this mechanism is also presented, along with some initial performance figures (e.g. 80ns for mobile communication on an 800 MHz Pentium 3).

With occam becoming available on a variety of microprocessors for GUI building, internet services and small-memory-footprint embedded products, these capabilities are timely. Lessons are drawn for concurrency back in OO languages - and especially for the JCSP (*CSP for Java*) package library.

1 Introduction and Motivation

Classical occam[1] has a *copy* semantics – data is copied from the sender to the receiver at the point of synchronisation. In contrast, communication in JCSP [2, 3, 4, 5, 6] goes with the flow of Java and has a *reference* semantics – only object references are sent. The same *channel synchronisation* semantics of CSP [7, 8, 9] applies to both occam and JCSP, but the after-effects are different. In occam, the sender and receiver hold separate *copies* of the communicated data – subsequent work by both processes on their respective data objects causes no mutual interference. In JCSP, the sender and receiver hold separate, but identical, *references* to the same object (which resides on the *shared heap* and hasn't actually moved anywhere) – this time, subsequent work by both on that object is a race hazard *if either of them updates any part of it*.

In summary, the occam communication is secure, but expensive if the data being sent is large. The JCSP communication is cheap (unit time cost regardless of data size), but secure only if the system designer stays on guard against the concurrent aliasing problem and doesn't make any mistakes.

For this research, rather than fight the culture of OO languages (where free-wheeling duplication of object references is the norm), we found it worthwhile to turn again to occam (where aliasing is at all times controlled) and see what can be done to ease or eliminate the copying costs of communication.

Between processes distributed over distinct memory spaces, copying the data will be unavoidable. Between processes living within the same memory space, copying only the references is possible. The trick is to make both scenarios semantically compatible and the latter one semantically safe. And, of course, to do this as simply as possible (but no simpler – *occam's razor* [10]).

This paper considerably expands on an earlier version [11] with details on mobile storage management, dynamically sized mobiles, parameter passing, undefined usage checks and performance.

2 Mobiles

For efficiency reasons, mobiles have been widely used in the past *as a design pattern* by occam programmers – for example in packet routers and GUI services. Various security checks have to be overridden in order to compile it (which means that it was not strictly occam any more), but the performance gains have been felt sufficient to justify the risk.

Although not thought about in quite the same way, the *mobile design pattern* is widespread in much OO programming for communicating information between different parts of a system – often with objects repeatedly created, used only briefly and then dumped. For applications that cannot tolerate the construction overheads, garbage collection and memory fragmentation caused by this, explicit creation and management of reusable object pools is a common solution.

Our proposal binds this design pattern into the occam language, ensuring its correct and efficient implementation without compromising security.

2.1 Mobile Semantics

Consider the *copy* and *move* operations provided by operating systems for managing files. The former duplicates the file, placing it in a target directory under a (possibly) new name. The latter just moves the file to the target directory, possibly renaming it. A key factor in the semantics of the *move* [12] is that the original path/file name is no more.

Consider now the assignment statement. Its purpose is to change the state of its environment, which we can represent as a set of ordered pairs mapping variables (or *l-values*) into data values (or *r-values*). In occam, because of its zero-tolerance of aliasing, assignment semantics is what we expect:

$$(< x_0, v_0 >, < x_1, v_1 >, ...) \quad \texttt{"x0 := x1"} \quad (< x_0, v_1 >, < x_1, v_1 >, ...)$$

In all other languages with assignment, the situation is more complex – since the variable '`x0`' may be aliased to other variables and the values associated with those aliases will also have been changed to v_1.

Consider next a mobile assignment statement. Its semantics is different in one crucial place:

$$(< x_0, v_0 >, < x_1, v_1 >, ...) \quad \texttt{"x0 := x1"} \quad (< x_0, v_1 >, < x_1, ??>, ...)$$

The difference is that the value of the variable at the source of the assignment has become *undefined* – its value has *moved* to the target.

The semantics for mobile *communications* have to follow naturally from the semantics for mobile *assignment*. In occam, communication is just a distributed form of assignment – a value is computed in the sender process and assigned to a variable in the receiver process (after the two processes have synchronised). For example, if the above '`x0`' and '`x1`' variables were of type '`FOO`', then the above (copy) assignment has to be semantically equivalent to:

```
CHAN OF FOO c:
PAR
  c ! x1
  c ? x0
```

That implies a key property of mobile communications – *the value of the output variable becomes undefined*.

There is an argument as to what *undefined* should mean. One possibility is to introduce it as an extra value (perhaps with the name 'NULL') on the underlying type and allow programmers explicitly to test for it. Another is to leave the type alone and define *undefined* to mean *any* value of that type. This means that the state of a moved variable becomes the same as that of a declared, but uninitialised, variable of the same type – formally $\perp$ in the denotational semantics of occam [13, 14].

For semantic and pragmatic reasons, we have chosen the latter of these two possibilities. The first leaves us open to 'NULL-*pointer errors*' at run-time and a somewhat artificial decision to make as to whether to allow 'NULL' values to be assigned or communicated. The second gives us a semantics for mobile assignment and communication that is *strictly weaker* than that for copy assignment and communication – and we will take advantage of this presently (see the start of section 3). It also allows the highly efficient management of *fixed-size* mobiles (sections 3 and 4). The downside is the need to guard against accidental use of undefined values (see section 7) – although we note that we have always had this problem for uninitialised variables and (mostly) ignored it!

2.2 Mobile Syntax

We propose two new keywords for the language: a 'MOBILE' qualifier for data types and a 'CLONE' prefix operator.

The MOBILE qualifier doesn't change the nature of the types *as types* – for example, MOBILE types are compatible with ordinary types in expressions and assignment. This is important, since we may wish to construct PROCedures, FUNCTIONs and operators [15] that will work when given variables of either type (see section 6).

However, it does impose the *mobile semantics* on assignment and communication between MOBILE variables. So, if we have the following declarations:

```
DATA TYPE FOO
  MOBILE RECORD
    ...  fields
:

FOO x0, x1:
```

then the assignment and communication code fragments in the last section have the *mobile*, and not the usual *copy*, semantics.

The MOBILE qualifier need not be burnt into the type declaration – it can be associated just with particular variables. For example, the following is an alternative to the above:

```
DATA TYPE BAR
  RECORD
    ...  fields (same as FOO)
:

MOBILE BAR x0, x1:
```

In some cases, we may actually want copy semantics for mobile variables. For this purpose, a “CLONE” operator is provided. This generates a copy of a mobile on which we can then perform the required operation. For example, in:

```
SEQ
  x0 := CLONE x1
  c ! CLONE x1
```

both operations leave the value of x1 value unchanged (i.e. we are back to copy semantics). Indeed, without the first CLONE above, the last line would be unsafe (since the value of x1 would be undefined) – see section 7.

2.3 Mixed Mobiles and non-Mobiles

At present, we allow mobiles to be assigned to non-mobiles of the same underlying type (and vice-versa). The semantics reverts to copying. So, MOBILE BAR and BAR variables may be inter-assigned – though not, of course, FOO and BAR variables (type equivalence is based on *names* not *structure*).

To be consistent, we also allow mobiles to be communicated down channels carrying the underlying non-mobile type. The sent variables are copied and left unchanged.

However, we do not allow non-mobiles to be communicated across a channel carrying a compatible mobile type. That would require *either* a run-time penalty at the receiving end (which would have to detect whether the incoming data was mobile) *or* the creation of an anonymous mobile (and a copy into it). Neither seems very attractive. The need for mixed mobile assignment and communication is not apparent – so it may be neater just to ban it.

There is another form of mixing that is more useful and we do allow. occam PROTOCOLs define message structures and there is no reason why they should not have *mixed* MOBILE and non-MOBILE components. For example, if:

```
PROTOCOL MIXED IS FOO; BAR; FOO
CHAN OF MIXED mixed:
BAR y:
```

then:

```
mixed ! x0; y; x1
```

leaves x0 and x1 undefined – but y unchanged.

We had earlier toyed with the idea of having special symbols for mobile assignment (<-) and output (<!). Then, the use between mobiles of ordinary symbols (:= and !) would mean copy semantics and there would be no need for the CLONE operator. But that would remove the above flexibility for mixed messages.

3 Implementation of Mobiles

As mentioned in section 2.1, implementing mobile operations by copying is a perfectly legal mechanism. For efficiency, this is precisely how *small* mobiles (e.g. MOBILE REAL64s or any data type less than or equal to around 8 bytes) are managed – the compiler simply ignores the MOBILE qualifier on them. Copying is also used for communication of mobiles between processes occupying *different* memory spaces (virtual machines).

The interesting case is communication between processes in the *same* memory space and, of course, for mobile assignments.

3.1 Mobiles in the Same Memory Space

Mobile data cannot live in the workspace of the process that uses them – that workspace may be reused by another process running in SEQuence with it, or by other processes in any encapsulating ALTernative or conditional (IF and CASE processes). They have to persist in a *heap-like* space that is globally available. Unlike conventional heaps, we enforce *zero-aliasing* on its elements (the mobiles), *zero-time construction* costs (for most of them) and *zero-time garbage collection* costs (again for most and unit-time, per-element-gathered, for the rest).

The obvious scheme is used: mobile variables hold only pointers to their actual data. Those pointers, of course, will not be apparent or accessible to the programmer. Mobile assignment and communication requires the copying of those pointers – not the data. However, unlike OO languages, we are not going to allow this to set up any aliases.

The semantics chosen (section 2.1) avoids the concept of NULL values – they are unnecessary, a source of run-time error and require checking at run-time (or suitable handling after accessing data at an invalid address). Therefore, we ensure that mobile variables hold, at all times, *valid pointers* – although the data pointed at might be *undefined* (the problems of which are addressed in section 7).

Classical occam has constraints designed to meet security requirements for embedded systems operating within finite – sometimes very small – memory. Such constraints are highly relevant to modern applications. Forbidden are recursion, run-time computed parallel (PAR) replication counts and run-time array sizes. Sticking to these constraints enables some interesting optimisations, but going beyond them is not too horrible (section 5) – and it does not prevent our optimised management of *fixed size* mobiles.

For example, the total number of all mobile variables (or mobile fields, if we allow nested mobiles) that will become active in a system is discovered by the compiler – and this is not prevented by separate compilation of components. Assume the size of all the types underlying those mobiles is known. Then, *the total size needed for the mobile heap* can be exactly calculated. All space for mobile structures can be allocated and initialised before main system startup (section 4) – hence, zero runtime construction costs.

An early plan was to maintain free-lists of mobile nodes – one for each underlying type – within mobile space. When a mobile variable lost its data because it *received* a new mobile by assignment/communication or because it went out of scope, the lost data was added to the relevant free-list. When it lost data because it was the *source* of a mobile assignment/communication, it picked up some *undefined* material from the free-list. Both these operations would be unit time.

However, a much simpler idea emerged. The free-lists are not needed. Instead, mobile communication and assignment are implemented simply by swapping pointers between source and target variables.

Formally, the model seen by the user remains that the direction of data movement in assignment and communication is one-way, even though the implementation is two-way. This is important to allow the normal copying implementations referred to earlier for *small* mobiles and for communication across memory space boundaries.

If it turns out that *swapping* mobile assignments and communications are a useful paradigm in their own right, we shall consider providing them as primitive operations. Their implementations will be trivial (albeit with extra costs incurred for small mobiles and distributed communications).

3.2 Mobile Assignment

We are modifying the KRoC[16, 17] compiler. This uses an extended transputer *ByteCode* (ETC) [18] as an intermediary, before generating native code. The *Transputer Virtual Machine* (TVM) has a simple stack architecture. So, the assignment:

```
x := y              -- for any MOBILE type
```

compiles to:

```
LD x                -- load 'x'
LD y                -- load 'y'
ST x                -- store in 'x'
ST y                -- store in 'y'
```

where what is actually being loaded and stored are *pointers* to the data. Simple and fast.

3.3 Mobile Communication

We could implement this using two channels and two communications – one in either direction. But that is expensive – two synchronisations instead of one. Instead, we have further extended the intermediate (transputer) *ByteCode* with two new instructions:

```
MIN                 -- mobile input
MOUT                -- mobile output
```

These instructions take pointers to the mobile variables (in effect, a pointer to a pointer) and swap them. Those arguments are pre-loaded on the TVM stack, along with the channel address, in the usual way. Even though they both effect the same operation, 'MOUT' needs to deal with the case that the inputting process is not committed (i.e. it is ALTing). The same algorithm used for the non-mobile channel output instruction (OUT) is safely reused.

3.4 Cloning

A CLONE operator on the RHS of an *assignment* turns the mobile operation into a *copy*. A CLONE operator on the RHS of an *output* introduces an anonymous MOBILE – with very local scope – into which the output MOBILE is *copied*. This is followed by *mobile* output from the anonymous variable. CLONE operators appearing anywhere else within an *expression* have no semantic effect (section 6) and are ignored by the compiler. The only other place where a CLONE operator has impact is on an argument passed to a formal MOBILE reference parameter of a PROC (also section 6).

4 Mobile Storage Allocation

Mobile data lives in *mobilespace* – along with *shadows* of all mobile variables. This applies to *fixed size* mobiles only – *dynamically sized* mobiles are discussed in section 5. Figure 1 shows an expanded declaration of FOO from section 2.2, along with the layout of *shadow* variables and pointers in *mobilespace*.

The compiler generates a static mapping of all mobile variables and data on to *mobilespace*. This is similar to how the (process) *workspace* and (array and record) *vectorspace* allocations are performed. For each declared (and anonymous) mobile variable, space is reserved in *mobilespace* for its initial mobile data, along with room for a pointer to that data – that pointer is the *shadow* of the mobile variable. For each instance of a PROCedure,

```
DATA TYPE FOO
  MOBILE RECORD
    [4]INT dest:
    MOBILE [32]BYTE payload:
:
FOO a, b:
SEQ
  ...
```

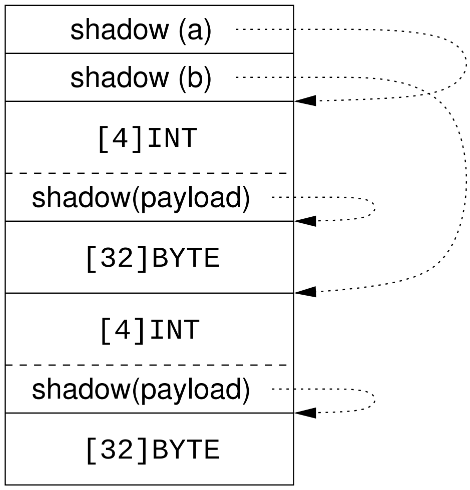


Figure 1: Example layout in mobile-space for "`FOO a,b`"

`FUNCTION` or operator requiring *mobilespace*, space is reserved in the caller's *mobilespace* – as is the case for *workspace* and *vectorspace*. The *mobilespace* requirements for a `PROC`, `FUNCTION` or operator are recorded in the output of the compiler, alongside the *workspace* and *vectorspace* usage.

In order for a `PROC`ess (`FUNCTION` or user-defined operator) to find its mobiles, an extra parameter is passed providing the address in *mobilespace* in which its mobile shadows live. The start of *mobilespace* is passed as a hidden parameter to the top-level process. This is the same mechanism already used to access *vectorspace* structures.

On initialisation, the KRoC run-time system allocates the whole of *mobilespace*, initialises it to '`MOSTNEG INT`', then passes it to the top-level `PROC`ess. In each *mobile-requiring* `PROC`, `FUNCTION` and operator generated, a special '`.MOBILESPACEINIT`' ETC [18] instruction is generated, which encodes the *mobilespace* map for that `PROC`/`FUNCTION`. This holds the *workspace* offset of the hidden *mobilespace* parameter (MSP), the number of mobile variables to initialise, then for each variable, the shadow-offset and the (initial) data-offset.

The translator turns this special ETC instruction into code to initialise that part of *mobilespace* the first time that `PROC`, `FUNCTION` or operator instance is called. The initialisation checks to see if the first word in *mobilespace* is '`MOSTNEG INT`', if it is, then the initialisation of mobile shadow variables (i.e. pointing them at their initial data blocks) is performed.

When a mobile variable comes into scope (i.e. at the point of its declaration), the pointer to its data is copied from its *shadow* word (already set up in *mobilespace*) into the process *workspace* word allocated for it. The compiler can statically determine the offset of a particular shadow variable from the hidden *mobilespace* parameter passed to the process. For the duration of the variable's lifetime, only this pointer now in the workspace is used (to save repeatedly loading from *mobilespace*).

When a mobile variable goes out of scope, the pointer it contains is copied back into its shadow in *mobilespace*. This may well be a different pointer to the one originally loaded (if the mobile variable has been assigned or communicated).

To keep the compiler generated *mobilespace* offsets low, the allocation strategy ensures that all shadow pointers for mobile variables in a particular process are allocated at the start of the (*mobilespace*) block addressed by the hidden parameter – these are followed by the actual data. *Mobilespace* for sub-processes are allocated below this. Of course, as execution and mobile operations proceed and mobile variables enter and leave scope, the mobile data areas *owned* by shadow variables migrate all over *mobilespace*.

4.1 Nested Mobiles

Nested mobile types (demonstrated by the FOO declaration in figure 1) present two problems.

Firstly, the data belonging to the mobile is no longer contiguous in memory (unlike the case for ordinary data types and non-nested mobiles). This has implications for the CLONE operator, which must now perform a *deep* copy. It also complicates the case when a mobile is output down a channel of its underlying (non-mobile) type. This has to be by a (contiguous block) copy since the receiving variable may be non-mobile (and contiguous) – see section 2.3. Space for a *serialised* version must have been allocated and serialisation performed. At least this will be simpler than the equivalent operation for Java (because the zero-aliased mobile structures can only be *trees*), but it is still not very pretty. This gives another reason to disallow such operations.

The second problem is sub-mobiles within non-mobile types. Suppose the BAR data type (section 2.2) is as described – i.e. it is a *non-mobile* that contains the same fields as FOO (which now includes a *mobile* `payload` – see figure 1). Non-mobile variables are normally allocated in *workspace* or *vectorspace* but, due to the recycling of these spaces by serial processes, we would lose the mobile field pointers (e.g. for `payload`). It would be possible to use a mobile shadow variable for each mobile field in a non-mobile type, but this would incur an $O(n)$ time cost (where n is the number of mobile fields) every time the variable entered or left scope.

For this reason we constrain any type with mobile components to be a mobile itself, and handle the $O(n)$ initialisation cost once when the enclosing 'PROC' or 'FUNCTION' is entered for the first time. This could be handled (secretly) by the compiler, but we prefer the programmer to tag explicitly such outer types as mobile – and generate a compiler error if they are not.

Finally, we note that nested mobiles cause no problem for mobile assignment and mobile communication within the same memory space. We still merely swap the top-level pointers – the lower-level ones need no adjustment.

5 Dynamically Sized Mobiles

So far, the mobiles presented have had statically allocated memory. But they have many of the characteristics of *heap* allocated objects (e.g. system-wide visibility and fast distribution via references) – except that construction never fails and is quick (just load the pointer from the shadow variable), there are no *null* states, no *null-pointer* errors, no aliasing problems and no garbage collection.

On systems with no memory constraints – such as those supported by virtual memory – one other kind of mobile becomes possible: the *runtime sized* array. These are much like other mobiles, except that they are allocated and freed dynamically, rather than kept in *mobilespace*. Dynamic mobile array types omit the dimension in their declaration. For example:

```
MOBILE []BYTE buffer:
INT n:
SEQ
  in ? n
  buffer := [n]BYTE
  ...  process using buffer
```

In their *undefined* state, dynamic mobile array variables are implemented to refer to zero sized arrays. Memory allocation is done with the use of a special *expression* that describes the quantity (and type) of memory to allocate. When the occam compiler cannot determine the size of an array at compile time (which is *usually* the case for dynamic mobiles) it inserts *run-time checks* to ensure that any accesses on the array happens within its bounds.

Note that although space for the '`buffer`' mobile is allocated by the above assignment, its elements are still *undefined*.

Unlike other mobile data-types, dynamic mobiles do not have a direct counterpart in the non-mobile world – i.e., we reject declarations of the form '`[]BYTE x`'.

5.1 Implementing Dynamic Mobiles

As mentioned previously, dynamic mobiles do not occupy *mobilespace*. Instead, two words are allocated in the process *workspace*. One for the pointer to the array, and one for the size (in *elements*) of the array. When the dynamic mobile comes into scope, its *size-slot* is set to zero – the pointer is left untouched (and possibly invalid). This avoids the problem of *nullness* since any use of an unallocated array will be caught by a run-time bounds check (or by a compile-time check – section 7).

To avoid large overheads in memory allocation, we use a version of the Brinch-Hansen algorithm for workspace allocation in parallel recursion [19], with *half-power-of-2-size-quantisation* of its free-lists (Wood [20]). To implement this dynamic memory management, two new ETC instructions were added, '`MALLOC`' and '`MRELEASE`'. There are only two places where dynamic memory allocation can occur – through the use of '`[n]BYTE`', or through the use of '`CLONE`'. In both of these cases, a new chunk of memory is pulled off the corresponding free-list – the run-time system will allocate more heap memory if it finds the free-list empty. In contrast, releasing memory to the free-lists can potentially happen in a number of places – before allocation, input and assignment. The generated (ETC) code for a dynamic mobile *check-and-free* sequence is:

```
  -- check-and-free code for 'var'

  LD (var + 1)      -- load array size
  CJ :skip          -- jump if zero, else
  LD var            --    load array pointer
  MRELEASE          --    free memory
:skip               -- program continues
```

Unlike ordinary (non-dynamic) mobile assignment and communication in the same memory space (section 3.1) we do not employ the same pointer-swapping technique – mainly because the source and target arrays may not be the same size. Instead, we revert back to the 'early plan' described in section 3.1 (but with the *half-power-of-2* sized free-lists).

5.1.1 Dynamic Mobile Assignment

For assignments involving mixed dynamic mobiles and non-dynamic mobiles, the assignment reverts to the default *copy* semantics. This is a fairly special case, and only works when the arrays are of equal size (the compiler will insert checks where necessary). For the other cases (where the LHS is a dynamic mobile), the code generated depends on what the RHS is. A dynamic mobile allocation of the form:

```
buffer := [n]BYTE
```

compiles to:

```
...  check-and-free code on ‘buffer’

LD n              -- load new size
LD <typesize>     -- load element size (constant)
PROD              -- multiply to get number of bytes
MALLOC            -- allocate memory
ST buffer         -- store pointer
LD n              -- load new size
ST (buffer + 1)   -- store in size-slot
```

This is a fairly generic version – the actual code generated may be quite a bit more complex, if ‘n’ is a FUNCTION call or a VALOF expression for example. The code generation for a dynamic mobile CLONE is slightly more complicated than the allocation code, for example:

```
thing := CLONE buffer
```

where ‘thing’ is of the same (dynamic mobile) type as ‘buffer’ would compile to:

```
...  check-and-free code on ‘buffer’

LD (buffer + 1)   -- load size of buffer
LD <typesize>     -- load element size (constant)
PROD              -- multiply to get number of bytes
MALLOC            -- allocate memory
ST thing          -- store pointer
LD (buffer + 1)   -- load size of buffer
ST (thing + 1)    -- store in thing’s size-slot

LD buffer         -- load source pointer
LD (buffer + 1)   -- load size of buffer
LD <typesize>     -- load element size (constant)
PROD              -- multiply to get number of bytes
LD thing          -- load dest pointer
REV               -- re-order top two stack elements
MOVE              -- copy data
```

The code-generation can get messy here since the *virtual transputer stack* is only three entries deep, hence the ‘REVerse’ instruction. The final special-case assignment is where one dynamic mobile is assigned directly to another – this is simpler (in time complexity) than the above two, for example:

```
thing := buffer
```

where ‘thing’ is of the same (dynamic mobile) type as ‘buffer’ would compile to:

```
...  check-and-free code on ‘thing’

LD buffer         -- load pointer
ST thing          -- store pointer
LD (buffer + 1)   -- load size
ST (thing + 1)    -- store size

LDC 0             -- load constant 0
ST (buffer + 1)   -- store in buffer’s size-slot
```

In all of these three cases, the compiler will attempt to avoid generating code where is safely can. The initial *check-and-free* can be avoided if we know that the array has a zero size at that point in the program. Additionally, for dynamic mobile BYTE arrays, the ‘LD <typesize>; PROD’ sequence can be omitted (‘<typesize>’ is 1). For sizes which are powers of 2, a shift-left instruction can be generated.

5.1.2 Dynamic Mobile Communication

Communication of dynamic mobiles is carried over channels of the dynamic mobile type. For example, the channel declaration:

```
CHAN OF MOBILE []INT c:
```

would be able to carry '`MOBILE []INT`' arrays. In addition to transferring the pointer between the sender and receiver, the size of the array must also be communicated. Ordinary mobile communication is handled by the '`MIN`' and '`MOUT`' instructions. We have added two more in line with this to handle dynamic mobile communication: '`MIN64`' and '`MOUT64`'. Unlike the non-dynamic pair, these instructions implement a 64-bit one-way transfer.

Before a dynamic mobile input is generated (for the inputting process), a *check-and-free* sequence is inserted to ensure that any previously held memory is returned to its free-list. Similarly, the *size-slot* in the outputting process is set to zero after the output. As with assignment, we avoid generating the *check-and-free* sequence if it is safe.

6 Mobile Parameters

Parameter passing is just *renaming* – at least, that is the formal position in occam. It is different to assignment and communication. So, there are no *mobile* semantic implications arising from this.

For instance, when we use mobiles within expressions (as function or user-defined operator arguments), we do not lose them. Recall that occam functions and operators are guaranteed free from side-effect – so there is no way they can communicate or assign from any mobile arguments we supply.

Initially, we implemented '`VAL MOBILE`' parameters, ensuring that the mobile variable is only ever read from – this involves extra checking for '`VAL MOBILE`'s on the RHS of mobile assignments and outputs. In doing this however, we lose the ability to exploit the *mobileness* of `VAL MOBILE` parameters, since any mobile assignment or communication of that parameter *must* use the `CLONE` operator. There is, therefore, no semantic point in having `VAL MOBILE` parameters and we ban them. Of course, a mobile variable can be passed to a `VAL` parameter of the underlying *non-mobile* type (the '`BAR`' type in section 2.2 for example).

Disallowing `VAL MOBILE` parameters also solves a problem of `FUNCTION`s that might take mobiles as arguments and return them as results. That would introduce aliasing.

Reference mobile variables passed to a `PROC`ess may, of course, be *moved* by that process (to another variable or down a channel). No problem. To ensure that any changed parameters are correctly restored, the compiler generates *copy-in, copy-out* type code. For dynamic mobiles, this includes *copy-in, copy-out* on hidden array dimension(s). It would be additionally beneficial to use the *copy-in, copy-out* parameter passing mechanism for small sized ($\leq$ 8 bytes – `INT`, `REAL64`, etc.) reference parameters, as this would avoid a lot of pointer dereferencing when using those parameters. Another option would be to shadow the parameter with a suitably typed variable allocated in the process *workspace* and employ a similar *copy-in, copy-out* strategy, but performed by the called process, rather than by the one invoking it.

Table 1 summarises the allowed formal and actual parameter-type combinations (where there are no '`VAL MOBILE THING`' formal parameters – see above).

In the cases where the actual parameter is a '`MOBILE THING`' and the formal parameter is not '`MOBILE`', we do not need to worry about handling nested mobiles (i.e. inside `THING`). The policy of nested-mobile typing (section 4.1) means that '`THING`' could not contain mobile sub-fields.

Dynamic mobile actuals follow the same parameter passing conventions as given by table 1.

	formal parameter		
actual parameter	`THING`	`VAL THING`	`MOBILE THING`
`THING`	yes	yes	no
`VAL THING`	no	yes	no
`MOBILE THING`	yes	yes	yes

Table 1: Summary of formal vs. actual parameter combinations

7 Undefined Usage Checks

A variable whose current data value (*r-value*) is *undefined* should never be used in that state. This is the initial state of occam variables – unless explicitly declared with `INITIAL` values [21]. We now have mobile assignment and communication that set their source variables back to this *undefined* state.

Checking against the use of uninitialised variables has been an omission from previous occam compilers. In the process of this research, we have added an *undefined* usage-checker to the (KRoC) occam compiler, that tracks the state of variables and channels through sequential code. Sequential channel usage checking is also included to catch code that would always result in deadlock – for example:

```
CHAN OF INT c:
INT x:
SEQ
  c ! 42      -- blocks here, waiting for
  c ? x       -- this input process
```

Essentially, at any point a variable is used, we determine it to be in one of three states: *defined*, *undefined*, or *unsure* (which means its *definedness* state depends on run-time happenings). For the most part, variables become *defined* when they appear on the LHS of an assignment, the RHS of an input, or as an actual parameter to a non-`VAL` formal.

For assignments, if any part of the RHS is *undefined* or *unsure* then the corresponding variable on the LHS is set to a similar state (*undefinedness* in expressions overrides *unsureness*). This extends to `FUNCTION`s and operators on the RHS of assignments, whose results are considered *undefined* or *unsure* based on the state of the actual parameters.

After mobile assignment or communication mobile variables become undefined. The compiler handles this correctly and will generate the appropriate warnings. For example, code such as:

```
PROC foo (CHAN OF MOBILE INT out)
  MOBILE INT x:
  SEQ
    x := 42
    out ! x
    out ! x
:

PROC bar (CHAN OF MOBILE INT out)
  MOBILE INT y:
  SEQ
    y := 42
    WHILE TRUE
      out ! y
:
```

generates the following compiler warnings:

```
Warning-oc-uc19.occ(6)- Variable ‘x’ is undefined here
Warning-oc-uc19.occ(14)- Variable ‘y’ might be undefined here
```

7.1 Implementing Undefined Usage Checks

The *undefinedness* checker is implemented as a separate stage in the compiler, which is performed after the alias and parallel-usage checking. For each variable[1] which comes into scope, a new ‘udv_t’ structure is created and added to a list which holds all the variables currently in scope. The key fields of this structure are:

```
struct {
    udv_t *next;                    // next in scope
    char state[];                   // state array
    treenode *nameof;               // pointer to symbol-table entry
} udv_t;
```

The state of the variable is recorded in the ‘state’ field, which is an array indicating the variable’s *definedness* at different points in the program. This ‘state’ array is treated as a stack, indexed by ‘udv_vstacklevel’, and is used to evaluate the state of variables inside nested code-blocks, such as IF guards, or the code within a WHILE loop.

Tracking the state of variables through branches of event guards and conditionals (ALTs, IFs and CASEs) is done by examining each branch, and its effects on the current state of variables at that time. ALTs require more careful handling than IFs and CASEs due to the possibility of declarations before the guard. For example, the code:

```
01    PROC dull (CHAN OF INT in, out)
02      ALT
03        INT x, y:
04        in ? x
05          out ! (x + y)
06    :
```

generates the appropriate warning:

```
Warning-oc-uc19a.occ(5)- Variable ‘y’ is undefined here
```

Figure 2 shows how the *undefinedness* checking for an IF conditional is performed. Just prior to the IF conditional, ‘n’ is defined and ‘u’ is undefined. As each guard is processed, the stack-level is incremented and the state prior to the IF copied. The body of the guard is then examined, leaving the resulting variable states at that stack-level. After the IF, a merge is performed which examines the output state of each guard with the initial input state (before the IF) and generates a resultant state. When the condition of the IF is found to be undefined, there is some argument as to whether analysis should be performed on the guarded process. The current situation is that this undefined analysis is performed for all code, regardless of any undefined variable usage leading up to it.

Checking parallel processes is done in a similar manner, collecting output states for each branch of the PAR, but the merge is different. Only one branch needs to change the state of a variable, and that will be carried through to the output state of the PAR. The *parallel usage* checker ensures that adherence to the CREW (*concurrent read, exclusive write*) model [22] is maintained.

Replicated SEQs and WHILE loops are handled by examining the loop body twice, along with the condition for the WHILE loop. As before, the stack-level is incremented and the state

[1] The use of ‘variable’ extends to *names*, which are not necessarily variables, in “VAL INT i IS (j \ n)” for example.

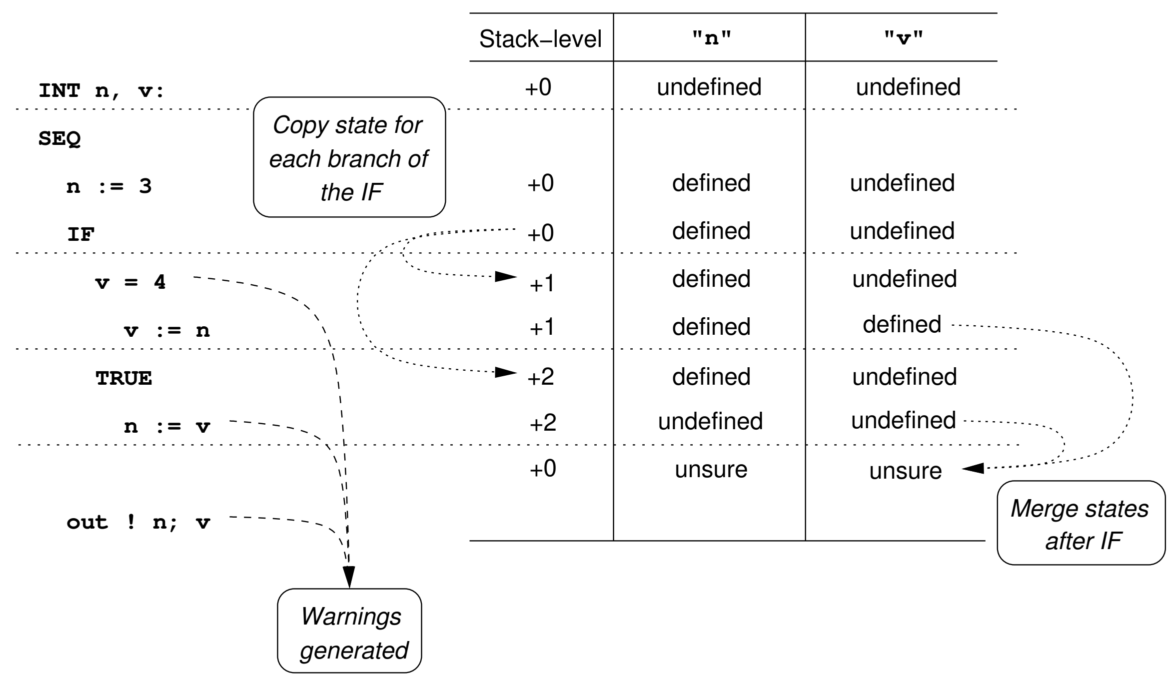


Figure 2: Undefined checking on an `IF` conditional

copied before the loop body is examined. After each check of the body, the states are merged back together. In this merge, *unsure* variables before the loop remain *unsure*. *Defined* or *undefined* variables change to *unsure* if their output state is different from the input state, otherwise they remain the same. This can be relaxed slightly if we know in advance how many iterations of the loop will be performed.

Replicated `IF`s, `ALT`s and `PAR`s are handled slightly differently, since there is no looping involved as such. For these, the replicated process is examined altering the current state directly – no merging after the replicator is needed. We can guarantee that the body executes exactly once for replicated `IF`s and `ALT`s, and possibly more times for a replicated `PAR`.

Value parameters must not be passed *undefined* or *unsure* variables. Reference parameters can be, however, since the invoked process may use it for returning a result. The explicit `RESULT` qualifier of occam3[21] would raise quality here, as we could then enforce all non-`RESULT` reference arguments to be *defined*. Additionally, we could ensure that `RESULT` parameters inside a `PROC` were left in a *defined* state when the `PROC` returns to the caller.

A more difficult issue is tracking the state of individual fields in array variables. In practice we treat arrays as atomic to keep the implementation relatively simple, although there is future scope for subscript analysis – the alias and usage checkers in the compiler go to great lengths to check array subscripts and slices. Analysing fields in record variables is not too difficult however – we just treat each <*variable, field*>-pair as an individual name.

For well-designed processes, these undefined usage checks will be straightforward. Compiler rejection (currently a warning) of the use of variables in *undefined* or *unsure* states – as well as partially *defined* arrays/records – will encourage better style. Warnings are not generated immediately, but collected and sorted first. This is to avoid repeated warnings, for example in "`n := (v + (v * v))`" where '`v`' is undefined, and to force them out in source-line order.

7.2 Undefined checks on nested PROCs

occam allows `PROC`edures and `FUNCTION`s to be declared wherever a normal declaration is allowed. This is generally a useful thing, but presents extra difficulty to the undefinedness checker. Take the following code for example:

```
01    PROC foo (CHAN OF MOBILE INT out, CHAN OF INT out.2)
02      MOBILE INT a:
03
04      PROC bar (VAL INT n, INT v)
05        SEQ
06          v := (n + a)
07      :
08
09      INT x:
10      SEQ
11        a := 42
12        bar (10, x)
13        out.2 ! x
14        out ! a
15        bar (10, x)
16        out.2 ! x
17    :
```

Here, the '`bar`' procedure uses the '`a`' variable, which is part of the '`foo`' procedure. Performing a simple undefinedness check on '`bar`' would lead to warnings being generated for '`a`', since it will always be considered to be *undefined* – it has just been declared. For this reason, nested `PROC`s are checked at the point of instantiation, and the state of the actual parameters followed through into the formal parameters. By checking nested `PROC`s this way, the undefinedness of '`a`' will be reported correctly. After examining the body of a nested `PROC`, the state of any non-`VAL` formals are copied back to the actuals. The compiler output for the above code is:

```
Warning-oc-uc20.occ(15)-     In call of 'bar':
Warning-oc-uc20.occ( 6)- Variable 'a' is undefined here
Warning-oc-uc20.occ(16)- Variable 'x' is undefined here
```

and no warnings are issued on lines 12 and 13 (where '`a`' is defined).

For top-level `PROC`s, we assume that any formal parameters are in the *defined* state when the `PROC` is called. However, within the same file, top-level `PROC`s are examined at their points of instantiation, as well as a normal "could be called from somewhere else" check.

7.3 Extra rules for mobile assignment and communication

The motivation, of course, for these undefined usage checks was the extra dangers introduced by our choice of mobile semantics. Whereas previously, variables could change state only once from initially *undefined* to *defined*, mobile variables may switch between these states any number of times.

However, this introduces no serious extra problems to the analysis described above. The whole code is checked in any case. We just have to record a mobile variable becoming *undefined* following its use on the RHS of a mobile assignment or output. This is trivial. No other changes to the analysis are required.

7.4 Note on the defined states of dynamic mobiles

Dynamic mobiles are treated slightly differently in the *undefinedness* checker. Any dynamic mobile array is considered *undefined* until it is allocated through the use of an '`[n]TYPE`' expression, or input from a (dynamic mobile) channel. For example, the following code generates an undefined variable warning for '`array`':

```
MOBILE []BYTE array:
SEQ
  array := "hello world!*n"
  ...
```

because '`array`' has had no space allocated yet.

However, we are thinking of allowing this assignment for the above example and other array-literals (i.e. tables), interpreting in the obvious way:

```
MOBILE []BYTE array:
SEQ
  VAL []BYTE tmp IS "hello world!*n":
  SEQ
    array := [SIZE tmp]BYTE
    array := tmp
    ...
```

Indeed, we are thinking of extending this to cover the general case of *non-mobile-to-mobile* assignment and communication. So, if '`x`' is a `MOBILE []THING` and '`y`' is a non-mobile `[`n`]THING` (where n is a known constant), then:

```
x := y
```

is compiled with:

```
SEQ
  x := [SIZE y]THING    -- dynamic mobile allocation
  x := y                -- copy assignment
```

8 Performance of Mobiles

Figure 3 shows the process networks of a producer-consumer and a ping-pong benchmark program, the results of which are shown in figures 4 and 5 respectively.

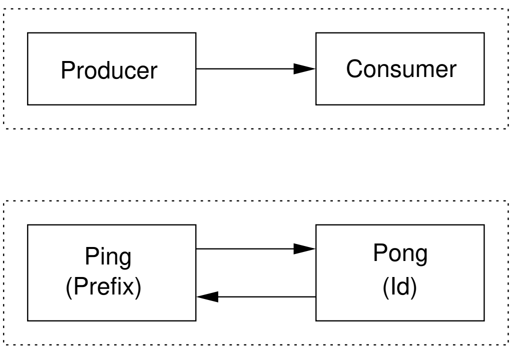


Figure 3: Benchmark process networks for producer/consumer and ping-pong

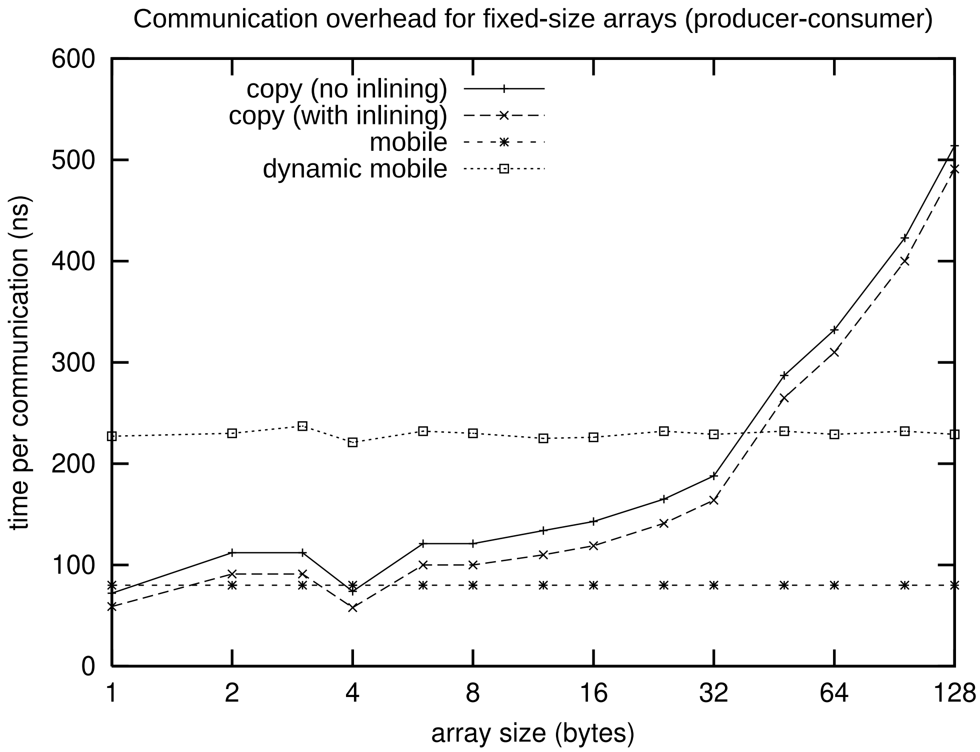


Figure 4: Producer-consumer communication overheads for different array sizes (measured on a P3 800 MHz)

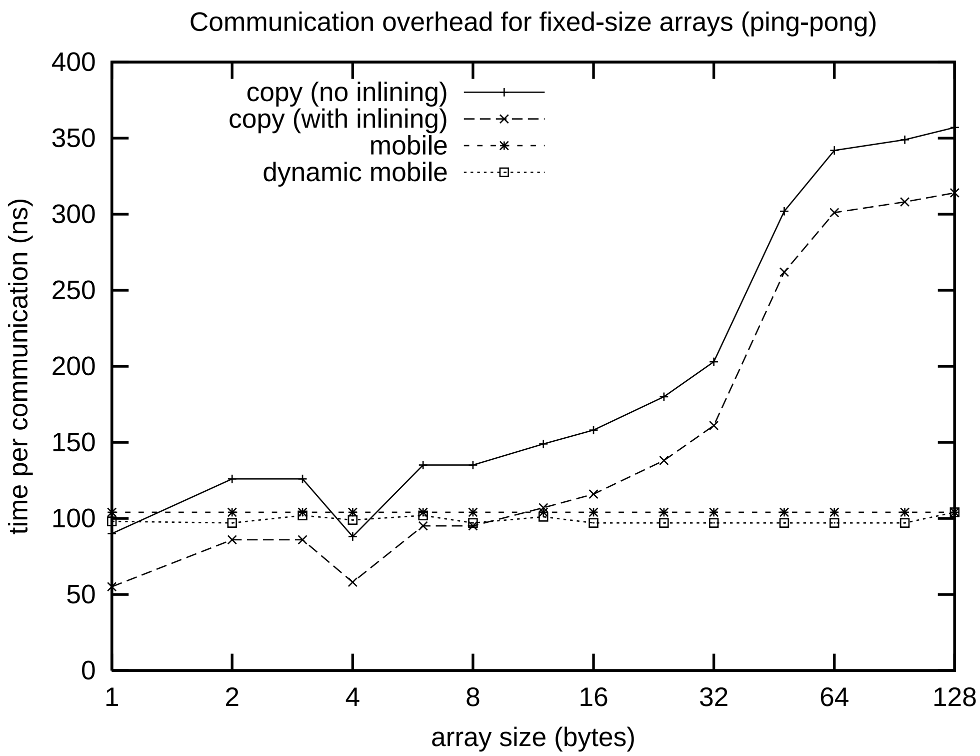


Figure 5: Ping-pong communication overheads for different array sizes (measured on a P3 800 MHz)

In the producer-consumer network, the producer repeatedly outputs a local variable, which the consumer repeatedly inputs. In the ping-pong network, the '*Prefix*' process outputs a variable then waits for input. The '*Id*' process is simply a buffer which performs an input then returns the variable to the '*Prefix*' process. An array of varying size (1 to 128 bytes) is communicated over the channel connecting the two processes. Two sets of results for *copy* communication are given, one is for an ordinary compile, the other is when *inlining* is enabled in the translator (`tranx86` [23]). Inlining here expands '`IN`' and '`OUT`' kernel calls for efficiency – as seen by the difference between the two curves. Inlining for mobile inputs and outputs has not yet been implemented, but should reduce the overheads of mobile communication by a similar amount.

For the producer-consumer network (figure 4) a local variable is communicated uninitialised (generating the appropriate *undefined* warnings – section 7), except in the dynamic mobile case, where it is allocated in the producer before being output to the consumer. Because of this, the dynamic mobile consumer process must perform a *check-and-free* on the target variable before the input. This accounts for the higher cost in communication for dynamic mobiles. However, as noted in section 5.1.1, we do save a *check-and-free* sequence before the allocation in the producer, since it is known that the array is always undefined at this point (it has either just been declared or output).

The ping-pong network (figure 5) produces slightly larger overheads for mobile communication (104ns compared to 80ns), but much lower overheads for dynamic mobiles (99ns compared to 230ns). The improvement in dynamic mobile performance is attributable to the removal of repeated allocations and *check-and-free* sequences, made possible by the undefined-variable checker (section 7). As can be seen in figure 5, dynamic mobiles experience slightly less overheads than static mobiles, showing the difference between a 32-bit swap (static mobiles) and a 64-bit one-way copy (dynamic mobiles). The general differences between the two graphs, where the producer-consumer network exhibits larger overheads for data copy than the ping-pong network, are attributable to processor caching effects. The significant improvement in the *inlined* copy is a feature of the inlining used, which reduces the amount of executed code considerably.

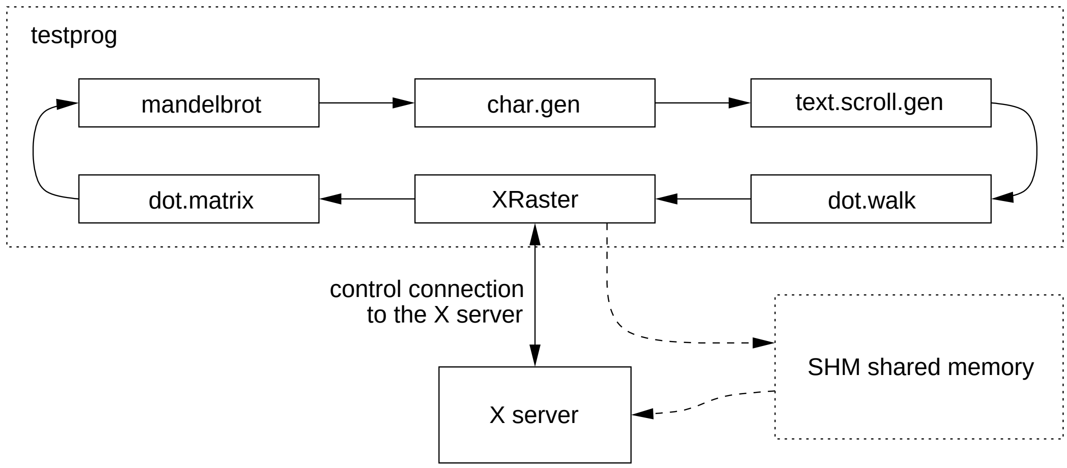
As expected, mobile communication has a roughly constant cost, regardless of the data size. The cost for allocating and freeing dynamic mobiles is slightly more variant, but mostly constant. This verifies that our free-list implementation for dynamic mobiles has a roughly constant overhead, regardless of the size of the data being allocated or freed. There is an initial (but hidden) cost initially to allocate a block of memory from the system, since the free-list will be empty when the first allocation happens.

Figures 4 and 5 show the times for the communication of data only – none of the processes involved access any array elements.

9 An Example of Using Mobiles

In order to test the implementation of mobiles, a small graphics library and test application has been written. The process network for the test program is shown in figure 6. The graphics library, called 'xraster', implements the minimal functionality required to get a bitmap from occam onto the user's X desktop. For this we have used MIT-SHM (MIT Shared Memory Extension) [24] – this provides a *memory-to-memory* copy for an '`XImage`' (one method X uses to represent bitmaps inside client applications), as opposed to sending the bitmap data across the X server connection. Of course, this only works when the client and the X server are running on the same same (System V IPC supported) system.

The library defines a mobile type, '`RASTER`', and a `PROC`ess to do the hard-work, using the declarations shown in figure 6.




```
DATA TYPE RASTER IS MOBILE [200][320][4]BYTE:
PROC XRaster (DISPLAY disp, VAL INT fps, CHAN OF RASTER in, out)
```

Figure 6: Process network for the MOBILEs test application

The 'fps' parameter to 'XRaster' specifies the speed (in frames-per-second) of the display. The connection to the X server is specified by the 'disp' parameter, which is obtained from another call in this graphics library. Internally, 'XRaster' calls the select() blocking system call (through a suitable interface in the C world [25]) to wait for an event from the X server or a timeout.

Initially, the 'XRaster' process outputs an undefined RASTER before entering its processing loop. When a timeout occurs, a RASTER is read from the 'in' channel and placed on the screen. The old RASTER is then sent down the 'out' channel. The various other processes in the network simply read a RASTER from their input, add some graphics, then send it on. The 'mandelbrot' process generates a real-time fractal zoom for a while, then continues generating the same image – using a CLONE.

In our test application, 'XRaster' is set to go at 20 frames-per-second. This results in a CPU load of around 1% (on an 800 MHz Pentium-III) after the mandelbrot zoom has finished. Increasing the frame rate to 50 frames-per-second results in a CPU load of $< 2\%$ after the zoom. Even at 200 frames-per-second, the CPU load is marginal, but the graphics hardware is incapable of displaying every frame. When we built the test application with a non-mobile RASTER, the CPU load was around 20% at 20 frames-per-second. Mobiles are a clear winner for this and other similar applications.

10 Conclusions and Future Work

Mobile communication *moves* data from the source process (which loses it) to the target. If source and target live in the *same* memory space, implementation is fast (just pointer swapping), secure (no aliasing is introduced) and consistent with communication between *different* memory spaces (ordinary copying). To nail the aliasing problem, the concept of mobile assignment has to be introduced – with complementary *movement* semantics and fast and secure implementation. The trick sought at the end of section 1 has been achieved.

Repeating this trick for existing OO languages (such as Java or C++) is not possible. We can get most of the semantics and fast implementation, but we cannot *automatically* control the aliasing and make it secure. This is the position of JCSP, where we rely on the user knowing the rules.

OO language change has to happen – essential concepts are missing. One of these is to separate, by good language engineering, the *different* uses to which pointers are put. They can still stay hidden (as in Java), but we must distinguish between their use for *sharing information* between different parts of the system (as in mobiles) and for building *interesting information structures* (such as doubly-linked lists). These ideas are discussed further in Tom Locke's paper to this conference [26].

The status of the occam (KRoC) work is that non-nested fixed-sized `MOBILE` types, variables, assignment, communication, parameter passing, extra usage checks, the storage allocation scheme (section 4) and the undefined usage checks (section 7) have been done. Dynamic mobiles have also been implemented, using free-lists and the Brinch-Hansen dynamic allocation scheme – section 5. Nested mobiles have not yet been fully implemented.

The KRoC compiler only recognises the extensions described in this experiment if a special flag ('`-X5`') is used. These extensions will be available in the forthcoming KRoC 1.3 release. We invite people to try using these `MOBILE`s and feed back their experiences to the community.

There is no space left to describe further applications. Whenever we have the pattern of accessing some data, processing it and passing it on, these ideas of `MOBILE` data are relevant – and that pattern is fairly prolific. Another example is the occam based web server [25, 27], where socket connections migrate from one end of the network to the other, having different operations performed on them by each process. The `MOBILE` qualifier was introduced on to the relevant data type and our experimental compiler produces a still working system! We haven't yet measured them, but the overheads *will* be lower.

We are also working on a full graphics/GUI library for occam, where almost all the communicated packets can be declared `MOBILE` and the load on the system significantly reduced. Currently, we are secure – but we copy.

The mobile pattern is endemic throughout OO systems (unconsciously) and most industrial scale applications of occam (sadly, from past years). But there is no automated secure management of that pattern and we have to take great care - and very often fail. This paper contributes to the automation of that care and a reduction in the cost of its execution.

References

[1] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: `http://wotug.ukc.ac.uk/parallel/occam/documentation/`.

[2] P.H.Welch and P.D.Austin. The JCSP (CSP for Java) Home Page, 1999. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`.

[3] P.H.Welch. Process Oriented Design for Java – Concurrency for All. In *PDPTA 2000*, volume 1, pages 51–57. CSREA Press, June 2000.

[4] D.Lea. *Concurrent Programming in Java (Second Edition): Design Principles and Patterns*. The Java Series. Addison-Wesley, 1999. section 4.5.

[5] P.H.Welch. Java Threads in the Light of occam/CSP. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 259–284, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press.

[6] P.H.Welch. Parallel and Distributed Computing in Education. In J.Palma et al., editor, *VECPAR'98*, volume 1573 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1998.

[7] C.A.R. Hoare. Communicating Sequential Processes. In *CACM*, volume 21-8, pages 666–677, August 1978.

[8] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[9] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

[10] Phil Gibbs and Sugihara Hiroshi. What is Occam's Razor? Available at: `http://www.weburbia.com/physics/occam.html`.

[11] F.R.M. Barnes and P.H. Welch. Mobile Data Types for Communicating Processes. In *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, June 2001.

[12] R. Strom and S. Yemini. The NIL distributed systems programming language: a status report. *j-SIGPLAN*, 20(5):36–44, May 1985.

[13] A.W. Roscoe M.H. Goldsmith and B.G.O. Scott. Denotational Semantics for occam2, Part 1. In *Transputer Communications*, volume 1 (2), pages 65–91. Wiley and Sons Ltd., UK, November 1993.

[14] A.W. Roscoe M.H. Goldsmith and B.G.O. Scott. Denotational Semantics for occam2, Part 2. In *Transputer Communications*, volume 2 (1), pages 25–67. Wiley and Sons Ltd., UK, March 1994.

[15] D.C.Wood and J.Moores. User-Defined Data Types and Operators in occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 121–146, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press.

[16] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, Concurrent Systems Engineering, pages 143–166. World occam and Transputer User Group, IOS Press, Netherlands, March 1996. ISBN: 90-5199-261-0.

[17] F.R.M.Barnes P.H.Welch, J.Moores and D.C.Wood. The KRoC Home Page, 2000. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`.

[18] M.D.Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 187–198, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press.

[19] Per Brinch Hansen. Efficient Parallel Recursion. volume 30 of *ACM SIGPLAN notices*, December 1995.

[20] D.C. Wood. An Experiment with Recursion in occam. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 193–204, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press.

[21] Geoff Barrett. occam 3 Reference Manual. Technical report, Inmos Limited, March 1992. Available at:
`http://wotug.ukc.ac.uk/parallel/occam/documentation/`.

[22] Peter H. Welch and David C. Wood. Higher Levels of Process Synchronisation. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 104–129, Amsterdam, The Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press. ISBN: 90-5199-336-6.

[23] F.R.M. Barnes. tranx86 – an Optimising ETC to IA32 Translator. In Majid Mirmehdi Alan Chalmers and Henk Muller, editors, *Communicating Process Architectures 2001*, Concurrent Systems Engineering, pages 265–282, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press.

[24] J. Corbet and K. Packard. The MIT Shared Memory Extension, 1991. Available at:
`http://pantransit.reptiles.org/prog/mit-shm.html`.

[25] F.R.M. Barnes. Blocking System Calls in KRoC/Linux. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures*, volume 58 of *Concurrent Systems Engineering*, pages 155–178, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press.

[26] T.S. Locke. Towards a Viable Alternative to OO – extending the occam/CSP programming model. In Majid Mirmehdi Alan Chalmers and Henk Muller, editors, *Communicating Process Architectures 2001*, Concurrent Systems Engineering, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press.

[27] F.R.M. Barnes. The occam Web-Server Home Page, 2000. Available at:
`http://wotug.ukc.ac.uk/ocweb/`.