

```

    { IF NULL (Alist)
      =>;
      $(Alist)<0
      => { Negative = Negative + 1;
          SET$ (Alist, -$(Alist))
          }
      $(Alist) == 0
      => Zeros = Zeros + 1
    }
    Total = Total + 1;
    OUT (Total, Zeros, Negative);
    Loop (&(Alist))
  };
PUBLIC { FUNCTION Extract < - Which;
  LOCAL Total, Zeros, Negative;
  RD (? Total, Zeros, Negative)
  { CASE < - Which;
    '+ => (Total-Zeros-Negative)/Total;
    '0=> Zeros/Total;
    '- => Negative/Total
  } };
  OUT (0,0,0);
  Loop (Alist)
}

```

After EXEC X = Absolute (Y), X.Extract would return one of three relevant percentages. The values are taken from a tuple in order to ensure consistency.

7. Speculative Processing

Speculative processing is a method to maximize the utilization of the processing capacity of a parallel system, by initiating tasks before knowing whether they will be actually needed, provided there is idle capacity in the system. If subsequently the results are indeed required, they may be immediately made

available, thus reducing waiting and overall elapsed time. A speculative task has to be purged, if during its execution, new information becomes available to indicate that its result will not be needed. There are also issues like how to attach the appropriate execution priority to a speculative task based on the likelihood of its usefulness, and adjusting this with changing execution conditions. It is desirable to do this without requiring application programmers to manage task status and priority directly because the expected level of system expertise is usually lacking.

In an earlier paper [10], we studied parallelism within conditional statements as the route to speculative processing: the spawning of THEN and ELSE modules with their Boolean guards in parallel, assigned priorities that decrease with their depth of nesting in IF statements and adjusted as Boolean results become available, provides a simple way to introduce speculative processing into application programs, one which does not require the programmer to get involved in system programming and task control mechanisms.

To preserve the semantic integrity of the program, the side effects of a speculative module must be confined to its private data space until it is confirmed by the Boolean conditional returns TRUE for a parallel THEN module or FALSE for an ELSE module. Unfortunately, this makes it impossible to decide whether to confirm or purge a speculative task by observing its own execution, and severely limits the usefulness of the speculative mechanism. The Boolean can only use external information to decide whether to confirm or purge a speculative task, not information from within the task itself.

For example, if we need the product of a number of factors that are being evaluated in parallel, and one factor turns out to be 0, then the evaluation of all remaining factors could be immediately abandoned. This could only be coded in our scheme as a set of nested IF..THEN.. blocks each returning one factor. A 0 result in one block causes a Boolean FALSE to be returned, purging all computation in inner blocks, but the outer blocks continue to completion, even though the result is already determined. The overall processing wastage may be reduced by putting factors most likely to be 0 in the outmost nestings, but this is not always easy to achieve.

While in terms of implementation, it is quite simple to provide a window through which the Boolean block can look into the THEN and ELSE modules, the problem is to do it in a semantically satisfactory way. Take the following example:

```

X = ...
Y = ...
IF {...}
THEN { X = ...;
      Z = ...
    }
ELSE { X = ...;
      Z = ...
    }

```

The THEN or ELSE block overwrites the old value X, and defines a new value Z, depending on whether the Boolean returns TRUE or FALSE, but while the Boolean block is still executing, it should see the old value X and undefined Z. If it is able to see the X and Z in the THEN and ELSE modules, then various questions arise that require the formulation of new scoping rules that allow the unique identification of the different items, e.g.,

- (a) how to specify a particular X out of the three;
- (b) how to declare Z which is defined inside the speculative module, and when it becomes visible from the Boolean module;
- (c) whether a speculative module need to declare which variables it intends to change;
- (d) whether a speculative module can reuse a variable name (e.g., Y in the above program, which the speculative modules do not use) and then discard its value at completion even if the module is confirmed, without impacting the main thread at all.

Reluctance to introduce such new and potentially untidy language rules has led us to leave the problem unsolved up to now. The root cause of the difficulty of tackling this apparently simple problem is that the speculative subspaces and their contents are creations of the runtime system and do not correspond to generic structures that have meanings across different application languages.

The situation changed with the introduction of the idea of active objects [11, 12]. Consider the construct

```
X = {IF boolean => EXEC A ( )}
```

The IF function is meant to return a pointer for an object of class A if boolean returns TRUE, but since the object execution is proceeding in parallel, a

pointer is immediately returned to the main thread as it executes boolean. This pointer is not visible outside the IF function, because boolean has not returned TRUE/FALSE and the assignment to X is not yet confirmed, but it is visible within the IF function as a speculative object pointer. The boolean is therefore able to obtain the public content of the speculative object using the unconfirmed but internally visible pointer X.

The use of a speculative *object* instead of a *function* eliminates the various scoping problems mentioned earlier, because an object call returns just one pointer to the main thread, with no other impact. Any new variables defined by the speculative module are encapsulated inside the object, and only the public parts of the information are visible from the outside. No new scoping rules need to be introduced at all, and the idea is generally applicable since it demands no specific object structure, as long as it permits the attachment of a tuplespace.

For an example of how to use the speculative capability, take the multiplication of two factors:

```
{ CLASS A;
  PUBLIC IN..., OUT...;
  EXEC OUT (Factor (1));
  OUT (Factor (2));
  SYNCHRONIZE
}
...
X = { IF { X.IN (? Product);
        { IF Product == 0
            => NIL
          T => T
        } }
      => EXEC A
};
{ IF NOT (NULL (X))
  T => { X.IN (? Y);
        Product = Product * Y
      }
}
...
```

The class A allows public access to its tuple operations IN and OUT; thus the tuplespace of object X may be accessed from the outside using X.IN and X.OUT. Its body computes two factors in parallel and puts the values into the object tuplespace. After the Boolean module retrieves one value, it return NIL (FALSE) if the value is 0, thus purging the evaluation of the other value and returning NIL to X. If the value is non-zero, it confirms the computation of the other value. The main program retrieves it from X later and multiplies it to the previous value.

Several other examples of common speculative processing are:

- (a) First come, first served — two functions are spawn, and the first result returned is accepted, while the other task is purged:

```

{ CLASS A;
  PUBLIC IN..., OUT...;
  EXEC OUT (Function1);
  OUT (Function2);
  SYNCHRONIZE
}
...
X = { IF { X.IN (? Result);
        NIL
      }
      => EXEC A
};
...

```

- (b) One not good enough, try other — the first result is accepted if it is satisfactory; otherwise wait for the other:

```

{ CLASS A;
  PUBLIC IN..., OUT...;
  EXEC OUT (Function1);
  OUT (Function2);
  SYNCHRONIZE
}
...

```

```

X = { IF { X.IN (? Result);
        { IF Happy (Result)
            => NIL;
          T => T
        } }
    => EXEC A
};
{ IF NOT (NULL (X))
    => IN (? Result)
}
...

```

- (c) Choose the more promising — two early results are compared, and the task with the poorer result is terminated:

```

{ CLASS A;
    PUBLIC IN..., OUT...;
    OUT (Function1);
    Function2
}
...
EXEC X1 = { IF { X1.IN (? Result);
                OUT (1, Result);
                IN (2 ? Signal);
                { IF Signal == 'die'
                    => NIL;
                  T => T
                } }
            => EXEC A
};
X2 = { IF { X2.IN (? Result2);
            IN (1 ? Result1);
            { IF Result1 < Result2
                => { OUT (2, 'die');

```

```

                T
                }
            T => { OUT (2, NIL);
                NIL
            } }
            => EXEC B
        };
    SYNCHRONIZE
    ...

```

Note that in this example a total of four threads are needed to maintain two speculative objects: the two Booleans must have separate threads so that they can each return NIL or T to purge or confirm the objects they guard. If speculation within speculation is permitted, the number of threads proliferate. However, in a runtime system with efficient task and tuple management this should not produce substantial overheads, because the Boolean module threads suspend until the early result tuples become available, and use little resources. Also note that a SYNCHRONIZE is needed to make sure EXEC X1 = ... is finished and the object result is ready. A SYNCHRONIZE is by default present at the end of a Boolean module attached to a speculative THEN module so that the thread continues only when the whole conditional finishes.

(d) Deadbeats — if a task executes for too long, kill it:

```

{ CLASS A;
  PUBLIC Status = NIL;
  ...
  Status = T
}
...
X = { IF { DELAY;
        { IF NULL (X.Status)
          => NIL;
          T => T
        } }

```

```

=> EXEC A
};
...

```

Note this one does not use A's tuple space, because if A hangs, any thread waiting for its tuples would also hang so that it would not be able to return NIL to kill A. Also note that, like in example c., if a number of parallel tasks are to be monitored and possibly killed, then a separate, parallel Boolean guard is needed for each, because only then can each independently return a NIL to kill the attached speculative object.

8. Object as Function Families

We suggest another application of objects, as function families. Consider the example of statistical averages: given a set of values X_1 to X_n , we compute weighted mean M , variance V and standard deviation S , using the weights W_1 to W_n (it is assumed the weights add up to 1):

$$M = \sum_{i=1}^N W_i X_i$$

$$V = \sum_{i=1}^N W_i (X_i - M)^2$$

$$S = V^{1/2}$$

Thus, we have several functions that share a common set of parameters W , which are initialized for each run by calling the following class:

```

{ CLASS Averages
  < - N : INTEGER,
    W : ARRAY [1..N] OF REAL;
  PUBLIC
  { FUNCTION Mean : REAL
    < - X : ARRAY [1..N] OF REAL;
    { FOR Sum = 0.,

```