COMMAN
A COMMUNICATION ANALYZER FOR OCCAM 2

by

Brian J. d'Auriol and **Virendra C. Bhavsar**

TR94-086 June 1994

Submitted to the Transputer Communications Journal

April 1994

Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, N.B.
Canada E3B 5A3

Phone: (506) 453-4566
Fax: (506) 453-3566

TRANSPUTER COMMUNICATIONS,

COMMAN — A Communication Analyzer for Occam 2

BRIAN J. d'AURIOL

Faculty of Computer Science, University of New Brunswick, Fredericton, New Brunswick, Canada

VIRENDRA C. BHAVSAR

Faculty of Computer Science, University of New Brunswick, Fredericton, New Brunswick, Canada

SUMMARY

We present a parser for analyzing communication overheads in OCCAM 2 programs for transputers. The input to COMMAN is a source file containing PROTOCOL statements which are defined by an augmented OCCAM 2 grammar. The output of the parser is a report detailing the expected communication times for the given input protocols. We show that COMMAN reasonably predicts transputer communication times for a variety of modes including internal and external communication, using both sequential as well as variant protocols. The parser is implemented using *yacc* while the lexical scanner is implemented using *lex*, which are utilities available in the unix and other environments. We give details of the *lex* and *yacc* implementations including the tokens recognized, the detection of scope and the grammar production rules. The communication times are modeled by linear functions and are built into the *yacc* grammar production rules.

KEY WORDS Communication Analysis, OCCAM 2, parsing, Transputer, multicomputers, parallel processing

1 Introduction

Our goal is to analyze a given OCCAM 2 program or program segment and determine the communication times required by any input or output communication in that program segment. We are only concerned with communications between pairs of processes, either executing on the same transputer (internal) or on two adjacent transputers (external). This prediction of communication times can be used in the static analysis of program code for measurement or performance purposes. For example, we use a version of the protocol analyzing parser described in this paper to assist in our evaluations of generated systolic implementations given in [1,2].

All OCCAM 2 communication occurs via channels where each channel is connected to a matching input (?) and output (!) OCCAM 2 statement. OCCAM 2 requires that most channel communication be datatyped by the OCCAM 2 PROTOCOL statement. Such datatyping specifies the exact sequential order and datatype of the data being transmitted. Our approach is based upon analyzing the datatype

????–????/93/010003–13$11.50
Received
Revised

components in a PROTOCOL statement. In this way, we obtain information about the communication behavior of any channels so datatyped.

In this paper, we present a parser for an augmented subset of the OCCAM 2 language as specified in [3] which computes the expected communication times for defined OCCAM 2 sequential and variant protocols. The parser is specified as a *yacc* [4] grammar while a *lex* [4] scanner provides the token stream to the parser. We have modeled our own *yacc* grammar after the one developed by Polkinghorne in [5]. Our lexical analyzer however, differs from Polkinghorne's lexical analyzer in that we use *lex* instead of a handcrafted analyzer.

This paper is organized as follows. Section 2 describes the overview of COMMAN and gives specific details of its implementation. We present our results for selected experiments in Section 3 and concluding remarks are given in Section 4.

2 COMMAN

COMMAN is an OCCAM 2 protocol analyzing parser for the transputer. Its purpose is to analyze a set of OCCAM 2 PROTOCOL statements and predict the required transputer communication times for any OCCAM 2 channel which is datatyped by these PROTOCOL statements. COMMAN is composed of two main functional units as shown in Figure 1, a lexical analyzer and a grammar parser. Input to COMMAN is a file containing PROTOCOL statements in OCCAM 2 format (with one exception as described below) for which channels used in a program segment are datatyped. The predicted communication times for these protocol statements is output. We use *lex* to process the input file into a stream of tokens and *yacc* to correctly parse the inputs. The communication analysis model is incorporated into the *yacc* parser.

COMMAN evaluates the internal or external (as specified by the user) communication times for each PROTOCOL statement in the input file. Since transmission of data on a transputer occurs serially, the transmission of a single BYTE takes less time than the transmission of a single INT32. Through empirical testing, a set of linear functions have been found to model transputer communication. This communication model, combined with the parsing function, allows COMMAN to be a useful tool in the static analysis of OCCAM 2 programs.

The remainder of this section is devoted to describing details of COMMAN. Section 2.1 describes the methods used to model the communication times. Sections 2.2 and 2.3 discuss the lexical analyzing and parsing methods used while Section 2.4 describes our changes to the interface between lexical analyzer and parser.

2.1 Modeling Transputer Communication

Communication on the transputer is influenced by three factors: the instructions used for the communication, whether the communication occurs internally or externally, and the state of compiler generated code specific to the communication process (i.e. the compiler options specified). The link transmission speeds impact only when external communication occurs. We assume that the link transmission speeds are constant throughout our discussion.

At the assembler level, communication on the transputer is initiated using one of three instructions [6]: `out`, `outbyte` and `outword`. Though empirical observation, we have determined that the `outbyte` instruction is used whenever the communication consists of single `BYTE` or `BOOL` datatyped values while the `outword` instruction is used for `INT32` and `REAL32` datatyped values. The `out` instruction is used for all other datatypes including arrays of `BYTE`, `BOOL`, `INT32` and `REAL32`. Since this instruction is the more used of the three, we concentrate our modeling efforts with respect to its usage.

The `out` instruction outputs a message packet of the required length. For a primitive datatype (other then those mentioned above) the length of the transmission is the length of the associated datatype. For example, a `REAL64` value has a message length of eight bytes. The length of a non-primitive datatype (e.g. an array) is the number of elements multiplied by the length of the datatype. An example of this is the output of `[5]INT32` which has a message length of 20 bytes.

Both internal and external communication is accomplished using the same instructions [6, Sections 6.8 and 10.4]. However, very different hardware operations occur. From our viewpoint, we need not be concerned with these differences since we are interested in analyzing the communication at the program level. We apply the same modeling method to each mode of communication.

The effect of different compilation modes also impact upon our model. In particular, the y compiler option [7, Section 25.7] controls the inclusion of debugging code into channel communications. The absence of debugging code allows faster communication. While not directly affecting the communication code per se, we group three additional options (k, u, a [7, Section 25.2]) together and consider the impact upon the model in two cases: unoptimized and optimized (where optimization in this case refers to the effects of these four options only). Similar to the effect of internal and external communication, we apply the same modeling method to each of these two cases.

We have conducted timing experiments on a multiple-transputer system to determine the communication times for various lengths of transmission packets. The transputer machine used in these experiments consists of one T800 - 25MHz transputer and eight T800 - 20MHz transputers on a B008 link board. Only one and two transputers were required for these timings, specifically one 25MHz and one 20MHz transputer. Selected timing results are shown in Figures 2 and 3 for communications of different array lengths consisting of the `REAL32` datatype. Actual timings were conducted with an iteration count of 1000 for accuracy reasons. Figure 2 shows the result for unoptimized internal communication while Figure 3 shows the result for optimized external communication. Timings for the other possible modes, optimized internal and unoptimized external, follow very similar trends. In Figures 2 and 3, a straight line represents the communication time (time 1) required for increasing message lengths. The decreasing curve shows the time (time 2) required per 32-bit word. Consequently, throughput increases with increasing message length (up to the maximum allowed by the hardware).

We summarize in Table 1 the results of a linear regression analysis conducted for each of the four mode categories. We note that there are different linear constants (slope and intercept) for each mode. Since the R-squared statistic is very close to one, the linear regression model accurately predicts the communication times in this experiment. Given that communication times are predicted according to $y_0 = mx + b$, the throughput curve was obtained by dividing the time to effect communication by the message length, that is $y_1 = x/y_0$. By substituting for y_0, we obtain $y_1 = b(\frac{1}{x}) + m$.

The PROTOCOL statement in OCCAM 2 indicates the datatype of the information that is to be communicated. Additionally, the PROTOCOL statement also defines a sequential order on the communication where different packets are delimited by a semicolon. Each packet is encoded at the assembler level using one of the three instructions mentioned above. There is thus a one-to-one correspondence between a datatype (whether primitive or non-primitive) and the output of the packet. Since our communication model evaluates each message in the protocol, the summation of the predicted times for all messages in the protocol will give an estimate of the total communication time required:

$$\text{Expected time}_i = \sum_{j=1}^{N} f(\text{packet-length}_j) \tag{1}$$

where N refers to the number of message packets in the i^{th} communication and f is the appropriate linear function to be applied to each packet.

Datatypes encoded as `outbyte` instructions can be modeled by our method in the following way. The effect of the optimization inherent in the `outbyte` instruction can be modeled by introducing an 'attenuation' factor into Eq. (1). This will appropriately reduce the estimated communication time for each packet. For this case, the total estimated time is computed by

$$\text{Expected time}_i = \sum_{j=1}^{N} \gamma(f(\text{packet-length}_j)) \tag{2}$$

where N and f are defined as before. The γ function computes the 'attenuation' factor based on the curve $y_1 = b(\frac{1}{x}) + m$ where b and m are found by applying a linear regression analysis on the corresponding y_0 curve. Specifically, a regression analysis was applied to the points generated by the following method: the number of bytes in the message multiplied by the division of the measured by the predicted times.

2.2 Lexical Analysis

The lexical analysis is provided by *lex* [4]. *lex* is a utility available on a number of platforms which scans an input source file and breaks the input stream into a series of defined tokens. The lexemes are specified as regular expression patterns in a specification file which is input to *lex*. In addition to the regular expressions, the input specification file also allow program fragments to be attached to each recognized token. Consequently, the lexical analyzer can perform an action for each token recognized. In our case, we use *lex* under unix and the program fragments are in the C-language.

Tokenizing an OCCAM 2 PROTOCOL statement means recognizing string patterns for all lexemes within the context of the statement. We categorize these lexemes into four groups as follows:

1. Regular expression patterns corresponding to an identifier and integer
2. All primitive datatypes and all symbols used in the construction of non-primitive datatypes
3. Special lexemes consisting of scope delimiters, end-of-line and comments
4. Two special lexemes not defined in the standard OCCAM 2 grammar (discussed in detail in Section 2.3).

The indentation characteristic of OCCAM 2 to signify begin and end of scope complicates the lexical analysis. The scope in an OCCAM 2 program is determined by its current level of statement indentation relative to the statement indentation of the construct defining the current segment. Each indentation level can be identified by two 'white spaces'. The identification of scope in a lexical analyzer is made difficult for the following two reasons.

- The white space character (including two white spaces back-to-back) may occur in one of three contexts: as fillers between language tokens (i.e. as between two identifiers), as indentation markers of scope, or as indentation markers for line continuation. The latter is defined in context of the semantics of a grammar production rule, with the only restriction imposed on the lexical analyzer being that the indentation level of the line continuation exceeds the level for the current scope.
- Whereas the beginning of a new scope can be identified by counting appropriate numbers of groups of two white spaces, there is no identifiable token which indicates the end of the scope. Only *after* the lexical analyzer scans in a token, can the analyzer identify that one or more exdents have occurred. Figure 4 provides an example of this scanning problem. In the protocol labeled 'p4' (also 'p5'), while the `CASE` statement is preceded by one two-space token and the identifier, `tag1,` is preceded by two two-space tokens, only after the colon is scanned can the appropriate number of exdents be determined. In this example there are two exdents.

Both of these language characteristics lead to problems when defining the lexical analyzer and parser using *lex* and *yacc*. In our application, we solve the first of these problems by using the end-of-line character as a delimiter and by not allowing line continuation and solve the second by providing a *token buffer* interface between *lex* and *yacc* (discussed in detail in Section 2.4).

The end of the line character serves to distinguish between white spaces used as fillers and white spaces used for indents. Indents can be identified then, when two-white spaces occur after the end of the line and before any other token. We use a boolean variable to track when indentations can occur and an integer variable to track the number of two-white space tokens detected on that line of input.

We have developed *lex* specifications as shown in Figure 5. In this specification the two regular expression patterns corresponding to an identifier (`id`) and an integer (`integer`) are specified in the first part. Regular expressions corresponding to individual tokens for all primitives and non-primitives are given in the second part. Also included here is the recognition of the special lexemes. Program fragments attached to each recognized token export the token identifier to the parser via the `return_proc` procedure not shown in Figure 5. Program fragments for processing certain tokens including the special lexemes differ in the required actions. An identifier is added to a symbol table by the `symlook` procedure not shown in Figure 5. The value of a recognized integer must also be retained by the `yylval` variable. OCCAM 2 comments are identified as starting with the double-hyphen and completing with the end of the line. The two variables required to identify the scope are included in the program fragment for the two white-space token shown at bottom of Figure 5. A list of all tokens recognized is given in Figure 6.

A number of auxiliary procedures called from the various program fragments complete the *lex* specification file. These procedures provide support for symbol table operations as well as passing tokens back to the parser (detailed in Section 2.4).

2.3 Parsing

The parser is provided by *yacc* [4]. *yacc* generates an LALR parser for the tokens which the lexical analyzer recognizes. While not requiring that *lex* be used as the lexical analyzer, *yacc* does allows a convenient interface to *lex* and consequently, is often used with *lex*. Similar to *lex*, *yacc* takes an input specification file which primarily specifies the language grammar in the form of production rules. Program fragments (again in our case, in the C-language) can also be attached to each production rule.

With one exception, we define a grammar for the `PROTOCOL` statement of OCCAM 2. This exception involves the counted array. In OCCAM 2, a statement of the form

```
channel ! 10::array
```

indicates that 10 elements from the array will be communicated. The declaration for this in the `PROTOCOL` statement might be

```
PROTOCOL name IS INT32::[]BYTE:
```

which indicates that one 32-bit integer is communicated followed by an unspecified number of bytes. In this example, the value 10 is output using the `outword` instruction followed by ten bytes output using the `out` instruction. Since our purpose is to evaluate a defined protocol, we require that the protocol specify the exact number of bytes involved in the communication.

We augment the language syntax so as to specify the length of the array as follows

```
length-variable(actual-length)::[]array-type.
```

The actual length reflects the user's expectation of what run-time conditions is expected. The corresponding new declaration for the above example would be

```
PROTOCOL newname IS INT32(10)::BYTE:
```

The full grammar in *yacc* form is given in Figure 7. A COMMAN program consists of one or more COMMAN `PROTOCOL` statements separated by one or more blank lines. The *yacc* production rule for a `definition` allows multiple `PROTOCOL` statements in either of its two forms to be included into a COMMAN program. There are three basic grammar elements to a `PROTOCOL` statement arranged in a loose hierarchy. A variant protocol consists of one or more taglists where each taglist is either an identifier or the combination of an identifier and a sequence of datatypes. The *yacc* production rule for `taglist` allows one or more taglists in either variant to be formed. The `simpleproto` production rule defines the syntax for sequences of primitive and non-primitive datatypes. Lastly, grammar production rules for the datatypes are `type`, `arrtype`, `primtype` and `primtype2`.

Incorporating the communication analysis procedure discussed in Section 2.1 into the *yacc* parser is a two step process which corresponds to implementing either Eq. (1) or Eq.(2). We must first evaluate the appropriate function for each defined packet after which we must accumulate the times for all packets involved in the `PROTOCOL` statement.

The grammar in Figure 7 has been devised in such a way that the recognition of primitive types is sufficient to classify individual packets; the semicolon is only used to define correct syntax. There are three possibilities for each packet: either it consists of a single primitive type, an array type, or a counted array type (where, as defined in above, the count is explicitly specified). We can succinctly evaluate the packet communication times by coding *yacc* action statements and attaching them to the appropriate primitive type production rules. Consequently, the actions associated with `primtype` represent the first step in the evaluation of the respective prediction equation (Eq.(1) or Eq. (2)) for primitive datatypes only. The actions associated with `primtype2` likewise represent the first step in the evaluation of the prediction equation, but for non-primitive datatypes. Note that the counted array consists of both a primitive and non-primitive component which is correctly modeled by our method. The second step in the evaluation of the prediction equation, namely the summation, is handled by the procedures `init1` and `init2` attached to the `definition` production rule, but not otherwise shown explicitly. These procedures also perform some housekeeping chores as well as formatting and printing the generated report.

2.4 *lex* and *yacc* Interface

The indentation problem in OCCAM 2 can succinctly be described as follows: The END token, representing an exdent or end of scope, is ***not explicitly*** given in the program source stream. Only when several conditions are met, including the recognition of some other token, can the END token be ***inferred.*** Furthermore, more than one END token may may be inferred at the same time. Since *lex* automatically scans the program source and returns the next token found, any END tokens which may be inferred will not be returned to the parser.

Our solution to this problem involves redefining the interface between *lex* and *yacc*. In the usual case, *yacc* (having generated the `y.tab.c` program) contains function calls to the *lex* generated function, `yylex`. When `yylex` recognizes a token, the corresponding token number is returned via the return statement to the parser. In our case, we need to *insert* appropriate numbers of the END token whenever the exdent is detected.

We use a *token buffer* as a queue and enqueue the required number of END tokens in this buffer. We place the currently recognized token as the last in the buffer. For consistency, we place *all* tokens recognized by the *lex* scanner into this token buffer. Normally, the buffer length will be just one (corresponding to the current token), however, the buffer will fill up whenever exdents are identified. We consequently, ignore the return value from `yylex`.

An intermediate procedure must now be defined between the parser and the lexical analyzer which will intercept calls to `yylex` (see Figure 8). The parser issues a request for a token to the intermediate procedure. The intermediate procedure checks the token buffer and returns the next token to the parser whenever the token buffer is non-empty. If the buffer is empty, then a call to the lexical analyzer `yylex` is made. The lexical analyzer places one or more tokens into the buffer and control returns to the intermediate procedure. At this point, the first token in the buffer is returned to the parser. Note that whenever the lexical analyzer recognizes a single token (normal operation) that token is returned (via buffer write and read) to the parser. Only when an exdent is detected, does the buffer fill up. In this case, the parser will retrieve the END tokens appropriately as if they were present in the input source file directly.

Two additional changes are required to the *generated* code from *lex* and *yacc*. Firstly, the generated code for the parser `y.tab.c` contains two calls to the lexical analyzer `yylex`. These calls must be changed so that the intermediate procedure is called instead. Secondly, when the end of file condition is detected by the lexical analyzer, the zero token is returned to the parser. The return of this special end of file token occurs within the generated code from *lex*. Consequently, this return statement must be changed so as to place the zero token into the token buffer.

We use the unix utilities *sed* and *awk* to make these changes. Since the changes need to be made after (but before the final application is linked) *lex* and *yacc* generates the lexical analyzer and parser, we include these additional utilities within our makefile program.

3 Results

In this section we compare the predicted values of communication times with those measured for two typical types of communication found in OCCAM 2 programs.

In the first example we consider communication as consisting of one or more bytes as follows

```
PROTOCOL 3byte IS BYTE;BYTE;BYTE:
```

which is a message of 3 bytes. Each packet (consisting of exactly one byte) is encoded by separate transputer `outbyte` instructions. Figure 9 compares the predictions for messages consisting from one to fifteen bytes for unoptimized internal communication. The top line corresponds to the prediction using Eq. (1) while the middle line corresponds to the prediction using Eq. (2). Note that the 'attenuation' model, Eq. (2), compares favorably with the actual measured times (bottom line).

Lastly, all time lines are linear as is expected. Thus the number of instructions directly correspond with the number of bytes in the message.

The second example consists of predicting more complex sequential and variant protocols. We have chosen five protocols given in Figure 4. Figures 10 and 11 compare the measured times (first column in each groups) with predictions using Eq. (1) and (2) for unoptimized and optimized internal communication respectively. Figures 12 and 13 present similar details for unoptimized and optimized external communication, respectively.

With respect to unoptimized internal communication (see Figure 10), COMMAN correctly predicts the communication times for the first four protocols to within 2% error using Eq. (1). However, there is an 8% error with respect to protocol 5. By employing the 'attenuation' factor, Eq. (2), the error for this protocol drops to less then 1%. COMMAN behaves similarly for the optimized case (Figure 11), although the errors range between 5%- 20% for predictions using Eq (1) and 1.5%- 15% for predictions using Eq (2).

The predictions, however, are not as good for the external communication transmissions. Predictions using Eq (1) have errors ranging between 13%-20% for the unoptimized case while errors range between 15%-20% for the optimized case. Obviously, predictions using Eq (2) fair worse.

Our prediction models rely on the linear properties of communication, that is, the transmission times increases linearly with the amount of communication (in bytes or words) that is involved. We have empirically derived the slope and intercept from measurements which involve *small* and *compact* code. By contrast, our comparisons in Figures 9 through 13 involve somewhat larger code segments (due to the more complex protocols involved). There is consequently, *additional* overhead involved with respect to the global communication between two points. We have investigated this further by concentrating on protocol 'p1'. The significant or dominant message packet is the array of REAL64 . When we measure the time for this component only using the program segment used for our comparisons, we find a 19.50% error. However, when measured using the program segment used to derive our constants, we arrive at virtually the same time as that predicted. This is due to the fact that we derive the constants from repeated measurements of arrays of this form — actually we have used arrays of REAL32 for our measurements, but the transputer encodes arrays of REAL64 in the same way using greater message lengths. We believe that the excess time in processing these messages then, is due to the larger code and variable space in use. Other experiments not relating to this paper also confirm this viewpoint.

4 Conclusion

We have presented an OCCAM 2 protocol communication analyzer, named COMMAN, for transputers. COMMAN reads, as input, a source file containing valid PROTOCOL statements and computes the expected times for data communications which use that PROTOCOL statement. The syntax of the PROTOCOL statement is defined by an augmented OCCAM 2 grammar. We have showed that our parser reasonably predicts the transputer communication times for various communication modes involving both sequential and variant protocols. Moreover, the prediction is more accurate for transmissions involving a small and compact code. However, our model is also useful for analyzing any typical OCCAM 2 code.

Our analysis of transputer communications indicate that the communication times can be predicted by a linear model. Both the type of communication (internal or external) and whether the compiler has optimized such communication impact upon the constants used in the model; the behavior remains invariant. We have derived constants specific for the T800 transputer on a B008 link board.

The language recognized by COMMAN includes all grammar elements, with one exception, of the standard OCCAM 2 PROTOCOL statement. However, the *counted array* datatype as specified by OCCAM 2 cannot be accommodated by COMMAN. Instead, the grammar for this datatype has been augmented to include a specified integer representing the expected length of that array. Changes to an OCCAM 2 source program may therefore be required before being analyzed by COMMAN.

We have successfully applied a version of COMMAN as discussed in this paper to the problem of evaluating the communication costs in OCCAM 2 implementation of systolic arrays. We use the results from our analyzing parser (in combination with additional analysis criteria) to predict the total execution time for these implementations.

REFERENCES

[1] B.J. d'Auriol and V.C. Bhavsar, 'Multi-Transputer Implementations of Systolic Algorithms: A Case Study', Technical report, Faculty of Computer Science, University of New Brunswick, P.O. Box 4400, Station A, Fredericton, N.B., E3B 5A3 (in preparation).

[2] B.J. d'Auriol and V.C.Bhavsar, 'Systolic and Wavefront Array Algorithms on Distributed Memory, Multiprocessor Computers', in *Proc. of Supercomputing Symposium '93-High Performance Computing: New Horizons (SS93)*, ed., L. Bauwens, pp. 47–54, Calgary, Alberta, Canada, (June 1993). University of Calgary, Calgary, Alberta.

[3] Inmos Limited, Prentice-Hall Ltd., Hertfordshire, UK, *Occam 2 Reference Manual*, inmos document no. 72 occ 45 01 edn., 1988.

[4] T. Mason and D. Brown, *lex & yacc*, O'Reilly & Associates, Inc., 1990.

[5] P. Polkinghorne, 'Occam 2 Lexical Scanner and *Yacc Parser*'. Software routines released into public domain, 1989.

[6] Inmos Limited, Prentice-Hall Ltd., Hertfordshire, UK, *Transputer Instruction Set - A Compiler Writer's Guide*, inmos document no. 72 trn 119 05 edn., 1988.

[7] Inmos Limited, Bristol, UK, *Occam 2 Toolset User Manual - Part 1*, inmos document no. 72 tds 275-02 edn., March 1991.

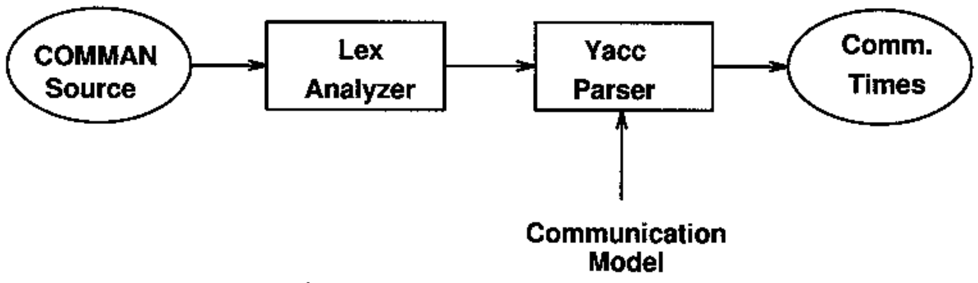

Figure 1. Overview of COMMAN.

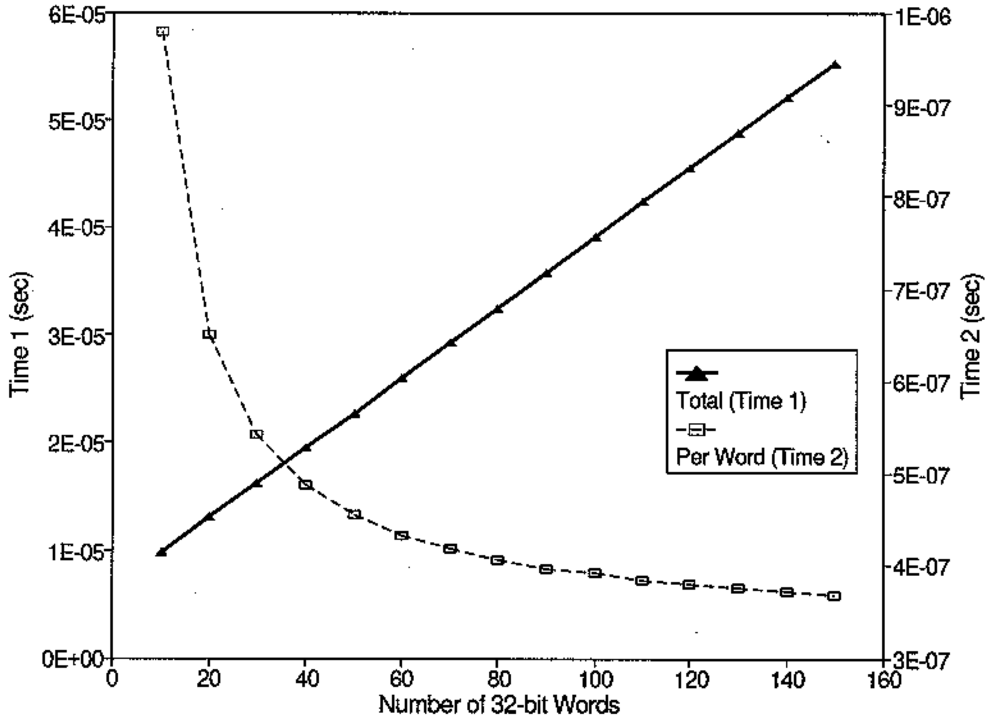

Figure 2. T800 transputer communication times for unoptimized internal communication.

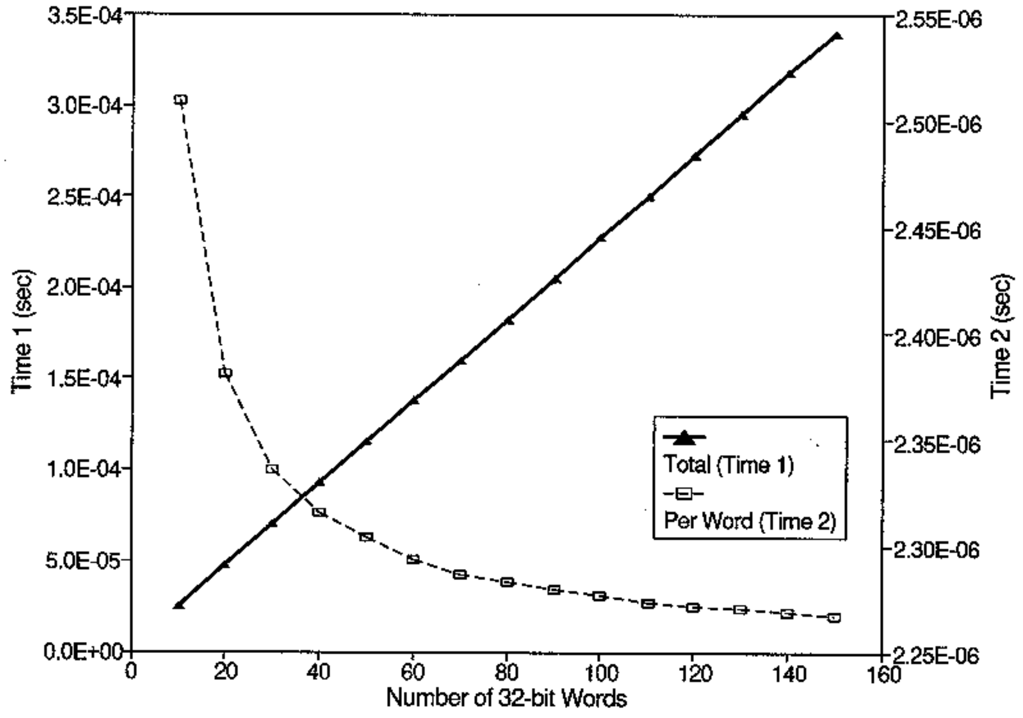

Figure 3. T800 transputer communication times for optimized external communication.

Table 1. Transputer communication linear regression model.

Communication Type		Regression Output		
Mode	Optimized	slope (m)	constant (b)	R squared
Internal	No	3.2466×10^{-7}	6.5333×10^{-6}	.99999
	Yes	3.2463×10^{-7}	3.0590×10^{-6}	.99999
External	No	2.2501×10^{-6}	4.6584×10^{-6}	.99999
	Yes	2.2501×10^{-6}	2.5862×10^{-6}	.99999

```
PROTOCOL p1 IS INT16;[24]REAL64;BYTE;[24]BOOL;INT32:
PROTOCOL p2 IS INT16;INT16(24)::[]REAL64;BYTE;INT32(24)::[]BOOL;INT32:
PROTOCOL p3 IS INT16;INT16(100)::[]REAL64;BYTE;INT32(100)::[]BOOL;INT32:
PROTOCOL p4
  CASE
    tag1;INT16;INT16(50)::[]REAL64;BYTE;INT32(50)::[]BOOL;INT32
    tag2
:
PROTOCOL p5
  CASE
    tag1;BYTE;[17]BYTE;REAL64;REAL32;BYTE(25)::[]REAL32;[10]INT16;BOOL
    tag2
:
```

Figure 4. OCCAM 2 protocols used in our experiments.

```
/* regular definitions */
digit           [0-9]
letter          [A-Za-z]
integer         {digit}*
id              {letter}({letter}|{digit}|\.)*

/* Token patterns and actions */
%%
PROTOCOL        {return_proc("PROTOCOL ",PROTOCOL);}
CASE            {return_proc("CASE ",   CASE);}
BYTE            {return_proc("BYTE ",   BYTE);}
BOOL            {return_proc("BOOL ",   BOOL);}
INT             {return_proc("INT ",    INT);}
INT16           {return_proc("INT16 ",  INT16);}
INT32           {return_proc("INT32 ",  INT32);}
INT64           {return_proc("INT64 ",  INT64);}
REAL32          {return_proc("REAL32 ", REAL32);}
REAL64          {return_proc("REAL64 ", REAL64);}
IS              {return_proc("IS ",     IS);}
";"             {return_proc("; ",      SEMICOLON);}
"::"            {return_proc(":: ",     COCOLON);}
":"             {return_proc(": ",      COLON);}
"["             {return_proc("[ ",      OPEN_SQBRACKET);}
"]"             {return_proc("] ",      CLOSE_SQBRACKET);}
"("             {return_proc("( ",      OPEN_RDBRACKET);}
")"             {return_proc(") ",      CLOSE_RDBRACKET);}

%{
/* identifier */
%}
{id}            {symlook(yytext); /*symprint(-1);*/
                 return_proc("id ", ID);
                }

%{
/* integer */
%}
{integer}       {yylval=atoi(yytext); return_proc("integer", INTEGER);}

%{
/* comments */
%}
"--"            {out("COMMENT\n");
                 while (  (c = getchar()) != '\n');
                }

%{
/* end of line */
%}
"\n"            {
                 return_proc("EOL\n", EOL);
                }

%{
/* indent/exdent tracking */
%}
"  "            {if (indent) {
                     current_level++;
                 }
                }

%{
/* ignore all other patterns */
%}
.               {}
%%
```

Figure 5. *lex* specification file.

```
%token          BYTE     BOOL     ID       EOL      PROTOCOL
%token          INT      INT16    INT32    INT64    REAL     REAL32   REAL64
%token          BEG      END      IS       CASE
%token          SEMICOLON         COLON    COCOLON

%token          OPEN_SQBRACKET    CLOSE_SQBRACKET INTEGER
%token          OPEN_RDBRACKET    CLOSE_RDBRACKET
%token          ERR
```

Figure 6. Tokens recognized by the parser.

```
program           :     sep definition
                  |     definition
                  ;

definition        :     PROTOCOL ID IS simpleproto COLON sep {init1();}
                  |     definition PROTOCOL ID IS simpleproto COLON sep {init1();}
                  |     PROTOCOL ID sep BEG CASE sep BEG taglist END END COLON
                                sep {init2();}
                  |     definition PROTOCOL ID sep BEG CASE sep BEG taglist END END
                        COLON sep {init2();}
                  ;

sep               :     EOL
                  |     sep EOL
                  ;

simpleproto       :     type
                  |     primtype OPEN_RDBRACKET INTEGER CLOSE_RDBRACKET COCOLON
                                OPEN_SQBRACKET CLOSE_SQBRACKET primtype2
                  |     simpleproto SEMICOLON type
                  |     simpleproto SEMICOLON primtype OPEN_RDBRACKET INTEGER
                                CLOSE_RDBRACKET COCOLON OPEN_SQBRACKET
                                CLOSE_SQBRACKET primtype2
                  ;

taglist           :     ID sep
                  |     ID SEMICOLON simpleproto sep
                  |     taglist ID sep
                  |     taglist ID SEMICOLON simpleproto sep
                  ;

type              :     primtype
                  |     arrtype
                  ;

primtype          :     BOOL    {xmit_time(0.25);}
                  |     BYTE    {xmit_time(0.25);}
                  |     INT     {xmit_time(1.00);}
                  |     INT16   {xmit_time(0.50);}
                  |     INT32   {xmit_time(1.00);}
                  |     INT64   {xmit_time(2.00);}
                  |     REAL32  {xmit_time(1.00);}
                  |     REAL64  {xmit_time(2.00);}
                  ;

arrtype           :     OPEN_SQBRACKET INTEGER CLOSE_SQBRACKET primtype2
                  ;

primtype2         :     BOOL    {xmit_time(0.25*yylval);}
                  |     BYTE    {xmit_time(0.25*yylval);}
                  |     INT     {xmit_time(1.00*yylval);}
                  |     INT16   {xmit_time(0.50*yylval);}
                  |     INT32   {xmit_time(1.00*yylval);}
                  |     INT64   {xmit_time(2.00*yylval);}
                  |     REAL32  {xmit_time(1.00*yylval);}
                  |     REAL64  {xmit_time(2.00*yylval);}
                  ;
```

Figure 7. *yacc* grammar specification file.

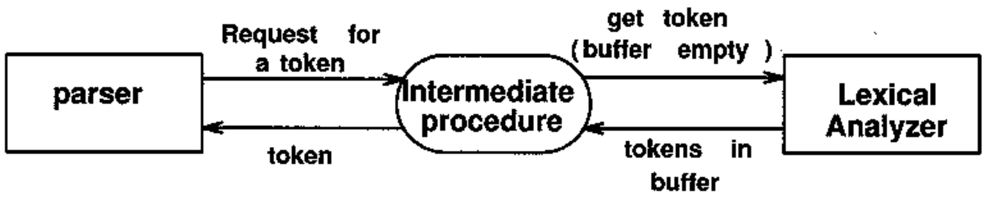

Figure 8. Interface between the parser and the lexical analyzer.

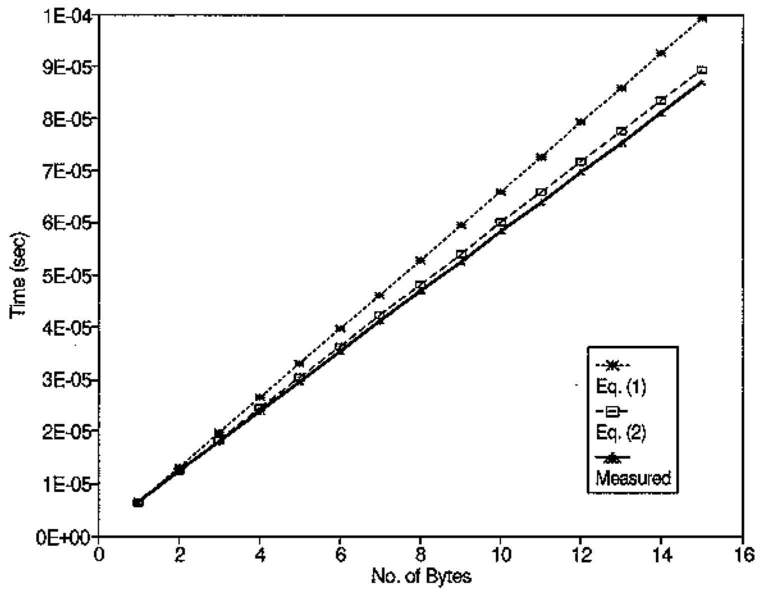

Figure 9. Unoptimized internal communication for messages consisting of bytes.

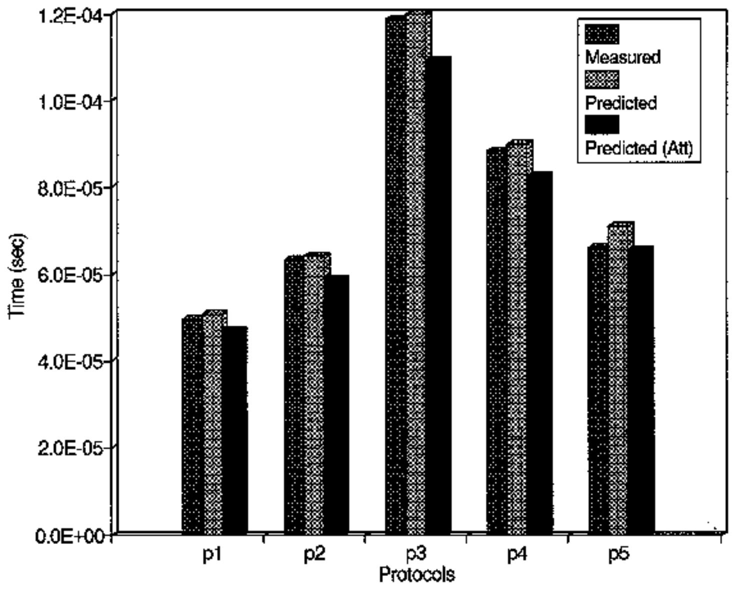

Figure 10. Unoptimized internal communication for various protocols

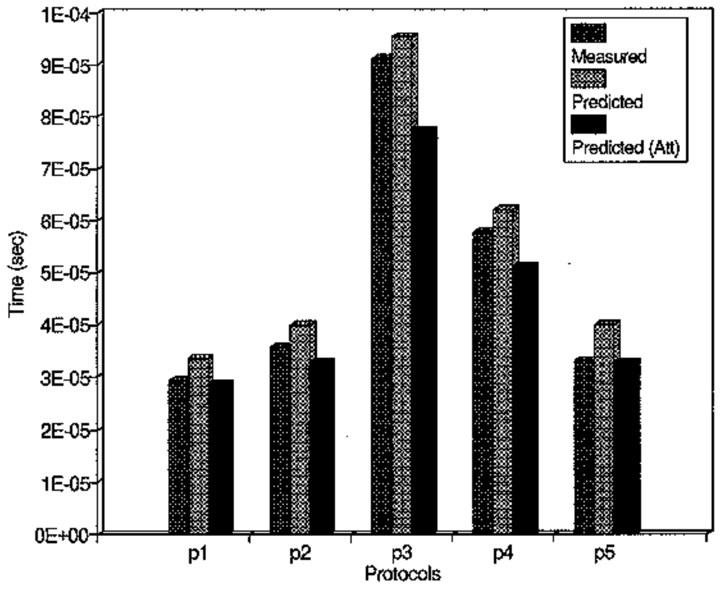

Figure 11. Optimized internal communication for various protocols

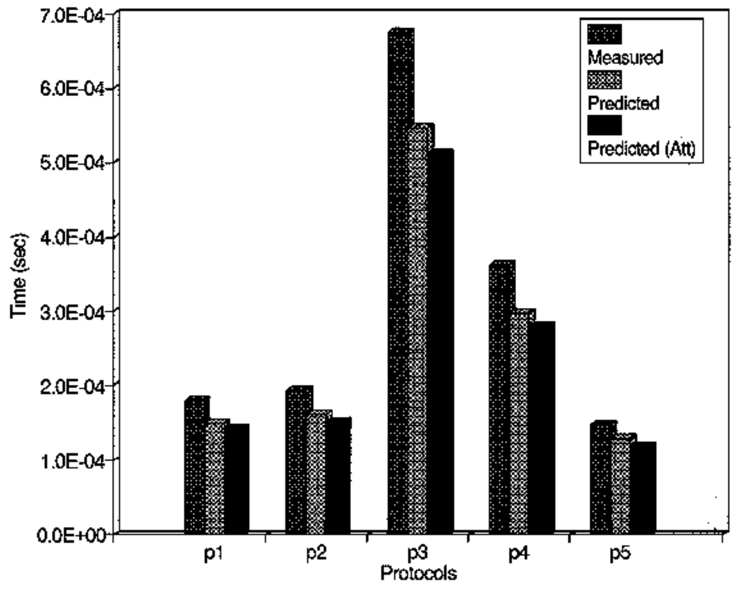

Figure 12. Unoptimized external communication for various protocols

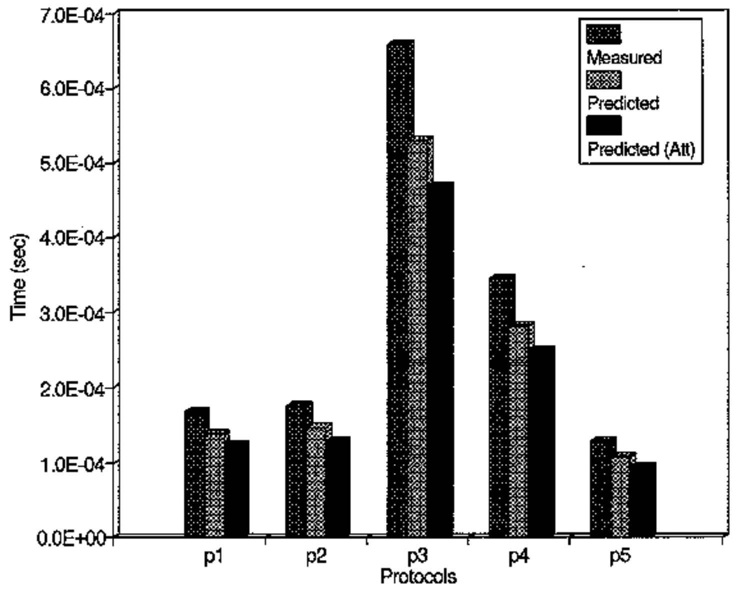

Figure 13. Optimized external communication for various protocols