

Millennium Team Project Design: Ephemeral compiler

A *System components*

Compiler front-end

This component parses Ephemeral code and performs machine-independent analysis. The output is an intermediate representation suitable for translation either to C++ or to MPP machine code messages. Preliminary semantic checking is done and violations will produce error messages. The front end will be incorporated into both compilers.

Mainframe compiler

This component produces a C++ translation of ephemeral code from the intermediate representation provided by the included compiler front-end.

MPP compiler

This component produces MPP machine code messages from the intermediate representation provided by the included compiler front-end.

MPP simulator

This component simulates operation of a target MPP, after accepting MPP machine code messages from the MPP compiler.

B *The Ephemeral language*

1 **Type system**

All data resides in the parameters of messages and is typed.

There are three categories of data types, and one other type category:

a *Bit strings*

These contain the ordinary data on which programs operate.

b *Code*

These parameters are actually bit strings which are sized by the compiler to contain action code specified in the program.

c *port references*

These act like ‘pointers’ to ports. And are used to send a message via the corresponding

port. General implementation of this feature for the 1D mesh MPP may be beyond the scope of this project. Instead, the contemplated implementation requires all port references in the MPP to have relative values which can be statically determined.

d Places

Types corresponding to places are used to indicate what message data is expected on each incoming port of a place, and which incoming message will be executed.

e Syntax

I attempt to use C++-like syntax wherever possible.

program ::= declaration*

declaration ::= action-declaration

 | bit-declaration

 | code-declaration

 | place-declaration

 | allocation-declaration

action-declaration ::= "action" name "(" "[name]" ")" "{" action-body "}" ";"

action-body ::= (message | allocation-declaration)*

message ::= port-expression ":" "(" expressions ")" ";"

expressions ::= expression | expression ";" expressions

expression ::= port-expression | a-expression | code-expression

 | a-expression "?" expression ":" expression // conditional

port-expression ::= name ":" name | name // placename.portname or port formal

a-expression ::= literal | name | "(" a-expression ")"

 | a-expression binary-op a-expression

 | unary-op a-expression

 // take account of operator precedence though

binary-op ::= "!" | "^" | "&" | "<" | "<=" | "==" | ">=" | ">" | "?=" | "+" | "-" | "*" | "/" | "%" | "<<" | ">>"

 // in approximate precedence order

unary-op ::= "-" | "~" | "!" // unary operators have highest precedence

```

literal ::= ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')* | ('0b'|'1b') ('0'|'1')*
           // non-negative decimal, or binary
code-expression ::= name // action name or code formal
allocation-declaration ::= name name "(" expressions ")" ";"
           // the first name is a placename
bit-declaration ::= "bit" name "[" a-expression "]" ";"
code-declaration ::= "code" name "{" name* "}" ";"
place-declaration ::= "place" name [ ":" ( "GCC" | "MPP" ) ] "{" portdecl* "}" ";"
portdecl ::= [ "<" ] name ":" "(" formals ")" ";"
formals ::= formal | formal ";" formals
formal ::= name name // typename name

```

2 Explanation

A program is a set of *declarations* where exactly one must be a `main` action declaration.

An action declaration is like a procedure declaration in C++, although more limited.

```
action actionname (placetype) { declarations body };
```

defines the *actionname* as an action with executable code generated from the *body*.

A *placetype* is used to indicate the type of place where this action may be executed, and therefore what message data it may access.

The *declarations* may contain place allocations, and possibly other types of declarations. Implementation of either of these kinds of declaration may be beyond the scope of this project. A place allocation creates (actually allocates from available resources) a new place, to which messages may be sent.

```
placetype newplacename [ (allocation-specification) ];
```

The optional *allocation-specification* allows the allocation of a specific place relative to the current place. For example the *allocation-specification* (22,-5) indicates the place that occurs 22 cycles after and 5 processors to the left of the current place.

The *body* comprises a collection of messages to be sent and ports to receive them.

```
Port : ( expression , ... ) ; ...
```

The *expression's* types must match the types in the port declarations of the relevant place type declaration.

Expressions of a port type may be a name of a formal port parameter from the place in which this action executes, or:

newplacename .portname

where *newplacename* refers to one of the new places allocated in the current action.

For each special architecture supported by the compiler, additional port names may be predefined. For example, the 1D mesh architecture may provide port names LEFT and RIGHT, respectively indicating the right incoming port of the processor to the left, and the left incoming port of the processor to the right.

Expressions of code types may be a name of a formal code parameter from the place in which this action executes, or may be the name of an action, in which case the action must have been listed in the declaration for the relevant code type.

Expressions of bit string type may refer to a formal parameter of bit string type, or may involve an arithmetic operation between other bit string expressions. Due to the short duration of an action, only very limited computations may be performed in an expression. We limit each expression to a single operation and also impose a limit on the total number of operations present in an action.

Expressions of any type may include a single selection operator:

(*condition ? trueresult : falseresult*) , which selects one of the results based on the condition.

The destination port for a message may be omitted; in that case, a new place of an appropriate new type will be allocated and the message will be sent to that new place. Implementation of this feature for the 1D mesh may be beyond the scope of this project.

The *expression's* types must match the types in the port declarations of the relevant place type declaration.

A bit string type is declared like an array of booleans in C:

```
bit typename[n];
```

A code type ultimately indicates a bit string large enough to hold code for actions designated by the programmer. It is declared similarly to a C union.

```
code codetypename { actionname , ... codetypename ... };
```

The compiler should guarantee that enough space will be reserved for this type to contain the code for any one of the actions named or any one of the other code types named.

Types for places define the ports and expected parameters, as well as which message is active.

```

place placetype : GCC {
    portname : ( typename formalname , ... ) ;
    -> portname : ( typename formalname , ... ) ;
    ...
};

```

This declares a new place-type, defining the names of its incoming ports, and the types of their formal message parameters. It also designates (with the arrow “->”) one special port as expecting an active message. The first formal parameter of the given port must be of a code type. The corresponding actual must contain the method code to be executed in places of this type. The implementation designator “GCC” indicates that this place must be implemented by compilation to C++. The default is “GCC”, in which case, the colon and the designator may be omitted. The designator “MPP” would indicate that this place must be implemented in MPP message code.

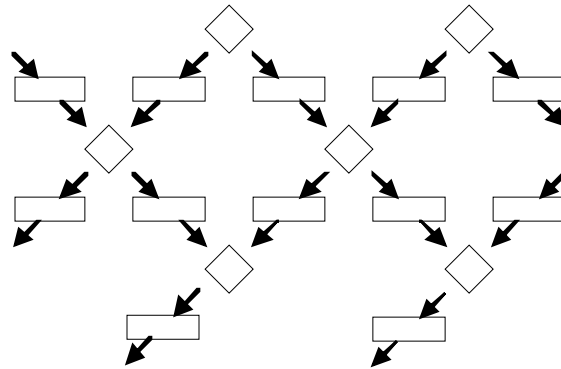
A port-reference type is indicated in a formal parameter list as: *placetype . portname* , which by reference to the *placetype* and *portname* indicates the type of message that may be sent via the port.

A port reference value must not be copied in an action. Within an action, only one mention may be made of a particular port reference value. An exception may be made in cases where conditional expressions are involved. This helps to ensure that the program will not attempt to use a port for more than one message.

C MPP instruction set

1 Topology

The MPP computer will comprise a linear array of virtual processors, each connected to its two nearest neighbor processors, possibly by shared message buffer. The processor on the “near” end is connected only to one nearest neighbor and to a “host” processor. The processor on the “far” end is connected only to one nearest neighbor. Each processor is active only on odd or on even clock cycles, depending on distance from the host. Processors with an odd distance from the host are active on odd cycles, and processors with an even distance from the host are active on even cycles. Each virtual processor retains no state from one cycle to the next. The state of the MPP is contained entirely in the contents of messages sent between odd and even processors. On its active cycle, each processor may receive a message from its left neighbor and from its right neighbor, and may then send a message to its left neighbor and to its right neighbor. The following pattern of processing activity is implemented in the array:



2 Message format

Messages comprise $128 + 8 \cdot 6 + 1$ bits. The first bit is 1 for an active message and 0 otherwise. The next $8 \cdot 6$ bits contain message formatting information and the final 128 bits contain message data. The message data comprises up to eight fields (numbered 0 .. 7) of up to 63 bits each. The message formatting information contains a list of lengths of each field. Bits $6n .. 6n+5$ of the format contains the length of field n . The final 128 bits are divided into fields according to the field lengths found in the message format.

Active	Format (48 bits)								Data (128 bits)					
	?	0	1	2	3	4	5	6	7	data 0 .. data 7				
	1	35	35	35	10	10	1	1	1	data 0 (35 bits)	data 1 (35 bits)	data 2 (35 bits)	3rd	4th

Format of Message

In this example, the message is active, the formatting is (35,35,35,10,10,1,1,1), and the data area is divided into 8 fields of 35,35,35,10,10,1,1 and 1 bits.

3 Instruction format

An instruction comprises thirty-five bits, further divided as two active-send bits, a 16-bit permutation operator and a 17-bit operation.

Instruction				
C	R	<table border="1" style="width: 100%;"> <tr> <td style="width: 50%; text-align: center;">permutation</td> <td style="width: 50%; text-align: center;">Operation</td> </tr> </table>	permutation	Operation
permutation	Operation			

C	R	P0	P1	P2	P3	OPCODE	S1	S2	Dest
---	---	----	----	----	----	--------	----	----	------

4 Active-send bits

These bits, further designated C and R, determine whether the sent messages will be active. C=1 makes active the message sent in the same direction as the generating message. R=1 makes active the message sent in the reverse direction from the generating message.

5 Permutation operator

The permutation operator comprises four permutation segments of four bits each. The permutation specified an arbitrary permutation of four groups of contiguous items, totaling 16 items. Each permutation segment specifies the starting position of one group of items. For example, the four segments (4,3,2,11) specifies four groups of items: items 4..10, item 3, item 1..2 and items 11..0 (wrapped around at 16). The ending position of items is given by the segment ordering. If the starting items are I(0) .. I(15), the example permutation reorders the items as follows: I(4)..I(10), I(3), I(2), I(11)..I(15),I(0).

The parameters from the active message are sources for items 0..7 for input to the permutation. The parameters from the other message (if it exists) are sources for items 8..15 of the permutation. The results of the permutations are available for use by the following operation.

6 Operations

An operation may be designated by the 5-bit OPCODE field, with sources and destination usually indicated by fields S1, S2 and Dest. The four-bit fields S1, S2, and Dest each indicate one of the message fields after permutation. The opcodes and meanings are described in the Operation Table. After the operation, the result fields 0 through 7 are grouped and sent as a message. This message travels through the array in the same direction as the active message. The result fields 8 through 15 are grouped and sent as a message traveling in the opposite direction. The result fields each retain the same number of bits as they had before the permutation, excepted as modified by the operation. Most operations determine the number of bits in the result based on the number of bits in the operands. Bit-level operations (OR,XOR,AND,SHR,SHRS,SHL,SHLS,INV,NEG) set the number of result bits to the length of the shortest input. Basic arithmetic operations (ADD,SUB) set the number of result bits to the length of the longest input. Special arithmetic operations (ADDX,SUBX) set the number of result bits to the one plus the length of the longest input. Many operations always produce a 1-bit result (LT,LE,EQ,GE,GT,DI,CZ). (continue after Opcode Table)

code	name	Description
0	NOOP	
1	OR	$S1 \mid S2 \rightarrow \text{Dest}$
2	XOR	$S1 \wedge S2 \rightarrow \text{Dest}$
3	AND	$S1 \& S2 \rightarrow \text{Dest}$
4	LT	$S1 < S2 \rightarrow \text{Dest}$
5	LE	$S1 \leq S2 \rightarrow \text{Dest}$
6	EQ	$S1 == S2 \rightarrow \text{Dest}$
7	GE	$S1 \geq S2 \rightarrow \text{Dest}$
8	GT	$S1 > S2 \rightarrow \text{Dest}$
9	DI	$S1 \neq S2 \rightarrow \text{Dest}$
10	ADD	$S1 + S2 \rightarrow \text{Dest}$
11	ADDX	$S1 + S2 \rightarrow \text{Dest}$
12	SUB	$S1 - S2 \rightarrow \text{Dest}$
13	SUBX	$S1 - S2 \rightarrow \text{Dest}$
14	MULL	$S1 * S2 \rightarrow \text{Dest}$
15	MULH	$S1 * S2 \rightarrow \text{Dest}$
16	MULF	$S1 * S2 \rightarrow \text{Dest}$
17	DIV	$S1 / S2 \rightarrow \text{Dest}$
18	MOD	$S1 \% S2 \rightarrow \text{Dest}$
19	SHL	$S1 \ll S2 \rightarrow \text{Dest}$
20	SHLS	$S1 \ll S2 \rightarrow \text{Dest}$ (signed)
21	SHR	$S1 \gg S2 \rightarrow \text{Dest}$
22	SHRS	$S1 \gg S2 \rightarrow \text{Dest}$ (signed)
23	CAT	$S1 \text{ concat } S2 \rightarrow \text{Dest}$
24	EXT	$\emptyset^2 \text{ concat } S1 \rightarrow \text{Dest}$
25	SHRI	shrink $S1$ by $P2 \rightarrow \text{Dest}$
26	COND	if (Dest) { $S1 \rightarrow S2$; };
27	CONDX	if (Dest) { exchange ($S1, S2$); };
28	INV	$\sim S1 \rightarrow \text{Dest}$
29	NEG	$-S1 \rightarrow \text{Dest}$
30	CZ	$!!S1 \rightarrow \text{Dest}$
31	COPY	$\text{Dest} \rightarrow S1, \text{Dest} \rightarrow S2$

Operation Table

The multiply-full (MULF) result length is the sum of the lengths of the inputs. The multiply-low, multiply-high and divide (MULL,MULH,DIV) instructions result length is the length of the first input. The modulo (MOD) instruction's result has the same length as its right operand (S2). The extend (EXT) instruction sign extends its left operand (S1) by the number of bits designated by the S2 field, placing the result in Dest. The shrink (SHRI) instruction removes bits from the left operand (by the number of bits designated by the S2 field) placing the result in Dest. The conditional move and conditional exchange instructions (COND,CONDX) have the same result sizes as input sizes. The copy instruction (COPY) operates in reverse parameter order, the destinations (S1,S2) will have the same number of bits as the source (Dest).

7 Message generation

After the operation, the result fields 0 through 7 are grouped and sent as a message. This message travels through the array in the same direction as the active generating message. The result fields 8 through 15 are grouped and sent as a message traveling in the opposite direction. The forty-eight bits of formatting information for each message are generated automatically according to the lengths of the result fields. The Active bit for each result message is generated according to the C and R bits of the instruction.

D Compiler front-end

The front-end of the compiler will be implemented in flex, bison, and additional C++ code to construct an intermediate representation and perform analysis.

1 Intermediate representation

The intermediate representation comprises the parse tree and various tables:

- The type table maps type names to internal representations of types.

- The code table maps code-names to sets of action names.

- The action table maps each action name to a list of generated places and a list of generated messages and their parameter expressions.

- The place table maps each place name to a list of message ports accepted by that place type, and lists of formal parameters with their types belonging to each message port

2 Machine-independent analysis

Ensure that all references refer to items that exist.

a Type checking for actual parameters in messages

Bit types

Ensure that only bit types are used in arithmetic operations.

Ensure that Bit fields are determined only by arithmetic expressions.

Code types

Ensure that Code fields will only contain values indicated in the relevant code declaration. Possibly allow overlapping code types.

Ensure that port references types match.

Ensure that first formal parameter of active message is of code type where all possible actions are allowed for the place receiving the message.

b Other

Check port references to prohibit copies and ensure receipt of messages. Analyzing conditional expressions is necessary, making this complex. Algorithm:

For each allocation declaration in each action,

For each port of the place corresponding to the declaration, check that the action contains exactly one port reference to that port. In the case of conditionals, two references in contradictory conditionals count as one

For each formal port reference parameter, check that the action contains exactly one usage of that port, either to send a message or as an actual port reference parameter.

Check allocation specifications to avoid place overlaps. Due to complexity, this may not always be decidable. The compiler will attempt to identify overlaps. Easily provable overlaps will be prohibited, while questionable situations may produce warnings.

E Mainframe compiler

1 I/O

Predefined place-types will be used to provide I/O capability for programs compiled on Linux. To do character output to the standard output, the users program creates a place of type output, and sends it a put message containing the character. To do input, the users program creates a place of type input, and sends it a get message. The get message is active and contains user code which will be given access to an input character, when it becomes available. These activities may be asynchronous, and the user must cause synchronization of different parts of a program if necessary. Non-blocking input is also available with place-type nwinput, which has an additional flag that indicates availability

of an input character.

The declarations:

```
place output {
    put:( int8 ch );
};

place input {
    ->get:( inputCode U ); // inputCode type must be user-defined
    data:( int8 ch );
};

place nwinput {
    ->get:( nwinputCode U ); // nwinputCode type must be user-defined
    data:( int1 avail, int8 ch );
};
```

2 Generation of C++ code

Code translation to C++ is relatively simple.

Each place type produces a struct type which contains internal structures corresponding to its message ports. Each formal parameter of a message port produces a field in the corresponding structure.

Each action produces a function declaration with a single parameter. The parameter is a reference to the place structure where it will execute. Each allocation in the action produces a "new" call to allocate a new place structure. Each parameter of each message in the action generates code to set the formal parameter fields of the internal structures of the receiving place structure. Each message generates a call to a method of the receiving place structure to indicate transmission of the message.

Code types and bit types exist only as abstractions and generate no code. All bit data is stored as long int data in C++.

Expressions in actions are translated recursively into C++ expressions in the obvious way, except that bit expressions may generate additional truncation code.

The struct's corresponding to place types will inherit from a common place structure which will provide mechanisms for queue management and scheduling of virtually concurrent actions.

F MPP compiler

1 Machine dependent analysis

a Location Analysis

Determine relative location of all places and port references.

Analysis must succeed, or program is considered erroneous.

Permits elimination of port reference parameters and allocation code.

Determines which direction corresponds to which ports.

Algorithm:

(not-optimized)

For each place declaration,

create a data structure (for decorations) with an entry corresponding to each pair of:

1. Any action which may execute in that place type
2. Any formal parameter, of type code or port reference type, occurring in that place

call this the decoration table

For each allocation declaration in each action,

Check that the declaration provides an allowable relative location in a time-like interval from the place of action.

Find all actual port reference parameters using the place named in that declaration and decorate them with the relative position from the allocation.

For each actual code parameter in the program,

if the code parameter is a reference to an action-name,
decorate the actual parameter with the action-name.

Iterate until no change:

For each active message in the program,

For each decoration (action-name) on the first actual parameter (the code) of the message,

For each decoration of each actual parameter in the message,

find the entry in the decoration table corresponding to the formal parameter and the action-name and:

if the parameter is a code type, take the union of the decorations on the actual parameter and the decorations in the decoration table, and store this in the decoration table.

If the parameter is a port reference,

if the decoration table entry is empty, copy the decoration of the actual parameter into the decoration table. (after adjusting for relative place)

Otherwise, compare the decoration of the actual parameter (after adjusting for relative place) with the decoration in the decoration table. If they don't match, declare the program ambiguous.

For each non-active message in the program,

For each action (action-name) which may execute in the destination place of that message,

For each decoration of each actual parameter in the message,

find the entry in the decoration table corresponding to the formal parameter and the action-name and:

if the parameter is a code type, take the union of the decorations on the actual parameter and the decorations in the decoration table, and store this in the decoration table.

If the parameter is a port reference,

if the decoration table entry is empty, copy the decoration of the actual parameter into the decoration table. (after adjusting for relative place)

Otherwise, compare the decoration of the actual parameter (after adjusting for relative place) with the decoration in the decoration table. If they don't match, declare the program ambiguous.

For each action in the program,

For each actual code parameter in that action,

For each formal parameter referenced in that code parameter,

Find the entry in the decoration table corresponding to that formal

parameter in this action.

decorate (union) the actual parameter with any decorations from that entry in the decoration table.

For each actual port reference parameter in that action,

Find entries in the decoration table corresponding to all formal parameters referenced in this actual parameter.

If this actual parameter is not decorated, but a table entry is decorated, copy the decoration table entry decoration to this actual parameter.

Compare all found entries with the actual parameter decoration. If any found decorations differ, declare the program ambiguous.

Check that each destination port of each message now has a decoration.

Check that this decoration indicates a destination place for the place of action.

In each action, check that all destination ports are to different places.

The relative placement of a destination will indicate which “physical port” is used.

If more than one destination used in a given action has the same relative location, an allocation conflict is indicated, and the program is considered erroneous.

b Allocation Conflict Analysis

Just as C code can violate the type system by constructing pointers to different objects at the same physical location, Ephemeral code can have allocation declarations which conflict with each other by allocating the same virtual place. Conflicts can not always be statically determined. The compiler will not attempt to perform advanced conflict analysis, deferring the detection of obscure conflicts to the simulator. Some simple conflicts are detected within the location analysis algorithm.

c Code generation Analysis

Check number of code and bit parameters in message and sum of field sizes for code and bit parameters. Port reference parameters have been eliminated. Capacity limits allowable programs. The arrangement of formal parameters and actual parameters must be analyzed to ensure that the permutation provided by the MPP instruction set is sufficient. An optional optimization could involve globally re-ordering message parameters to make the required permutations fit within what the MPP instruction set provides. This is likely to be beyond the scope of this project.

2 Code generation

Each action compiles as one instruction. Consequently, actions can contain only one

operation, and some movement of parameters. Each ephemeral message parameter corresponds to a message field in machine code. Sizes of arithmetic result fields must match those which can be produced as the result of instruction execution in the MPP.

The actual code generated will be the messages indicated in the ‘main’ action.

G MPP simulator

1 Inputs

The simulator will accept a sequence of messages as generated by the MPP compiler.

2 Data structure

The data structure representing the state of the simulated machine will be an array of messages being sent between processors.

3 Processing

The simulator will loop once for each simulated processor cycle. The user will determine (by a parameter) how many cycles to simulate.

On each main cycle, an input message may be entered into the leftmost simulated port.

Within each processing cycle, the simulator will loop over the array of processors, simulating each active virtual processor in turn.

4 Outputs

The user will be given the contents of all messages remaining in the simulator at the end of the simulation.