An Intensional Engine on L4

A Dissertation submitted to the School of Computer Science and Engineering of The University of New South Wales in partial fulfillment of the requirements for the degree of Bachelor of Engineering (Computer Engineering)

Vincent Jayawardene
(2172871)

October 1999

Supervisor: Gernot Heiser

Abstract

Intensional programming is a paradigm of programming that breaks away from the usual imperative computational model. An intensional program is based on intensional logic where the value of an expression depends on the context in which it is computed.

The most common implementation for intensional programming is a demand-driven computational system. Demands in an intensional system are similar to queries in a database system, except instead of requesting values, an intensional demand requests a value computed in a specified context. To facilitate this system an intensional engine is required to process demands and compute the results. The advantage to this system is that multiple demands can be computed in parallel.

However implementations of intensional engines suffer in efficiency because the overhead in propagating a large number of demands around the engine far outweighs any benefit gained from parallel execution.

The aim of this thesis is to implement an intensional engine on top of the L4 μ-kernel. As L4 has very low overhead for thread and inter process communication operations, the overhead in managing demands in the engine should be kept to a minimum.

The implementation of the L4 μ-kernel used for the engine only runs on a single-processor system which will prevents any parallel processing. This thesis is supposed to be a stepping stone for implementation of the engine on multi-processor versions of L4 or potentially on other distributed environments with very low communication overhead.

Acknowledgements

I would like to thank heaps of people (and a cat)... everyone who directly helped me in some way on this work, Gernot, Alan, John, Joey, the other guys in the K.E.G. Lab... my family, Liz, Lal and Amy for not letting me fall to bits near the end... That cat that lives in the driveway for always being in that driveway, even late at night... my friends Natalie, Abhinav, Kirsten, Genevieve, Mark, Elaine for being around when I needed to just blah, for telling me occasionally to go get some damn work done, and for just being good friends... my flatmate Rob for not whinging to much seeing as I havent done any cleaning in the flat for at least a month... And everyone else whos been around and I've momentarily forgotton...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Intensional programming is based on intensional logic, where the value of an expression depends on the context in which it is computed. Intensional programming allows the programmer to explicitly define the context of computation for expressions in a program.

Intensional programming is easiest in a language that easily shows intensional concepts. Currently the only general purpose intensional programming languages are variants of the Lucid[7] language.

The actual execution of an intensional program is usually carried out by a demand-driven computational system, or *intensional engine*. Demands in an intensional engine are similar to queries in a database system, except instead of requesting values, an intensional demand requests a value computed with respect to a specified context. The advantage to this system is that multiple demands can be computed in parallel.

However implementations of intensional engines suffer in efficiency because the overhead in propagating a large number of demands around the engine far outweigh any benefit gained from parallel execution.

There are two ways to deal with the excessive overhead in demand propagation. The first used by the GLU compiler[3] is to increase the granularity of demands by using large computation heavy functions to generate the values for demands.

The second method which will be attempted in this thesis is to reduce the communication overhead that forms a major part of the demand overhead. In order to reduce this overhead the engine will be implemented at a very low level. Essentially the only thing between the engine and the hardware will be the high performance L4 μ-kernel.

As L4 has very low overhead for thread and inter process communication operations, the overhead in managing demands in the engine should be kept to a minimum.

The aim of this thesis is to implement an intensional engine on top of the L4 μ-kernel on the MIPS-R4600[2]. This implementation will be called IE4[1] (Intensional Engine on L4). The IE4 system not only includes the engine itself but also the interface between the intensional programming language *Ilucid*, a variant of Lucid, and the IE4 engine.

1.2 Thesis Structure

The structure of this thesis is shown below:

Chapter 1 this introduction.

Chapter 2 introduces the basic concepts of intensional logic and intensional programming. It also introduces intensional engines in general, and two implementations of intensional systems.

Chapter 3 gives a very brief history of the Lucid intensional language and defines Ilucid, a variant of Lucid, the intensional programming language

[1] This is not an attempt to infringe on anyones copyright, it is merely an example of another parallel between intensional programming and the world wide web (see 2.3).

of the IE4 system.

Chapter 4 outlines the IE4 intensional system. It gives details of the conversion process between intensional Ilucid code and C code which will run on the IE4 engine. It also defines the IE4 client interface.

Chapter 5 outlines the services provided by the L4 μ-kernel and iOS the underlying operating system that IE4 is built on.

Chapter 6 outlines some design issues in the IE4 engine itself, then gives details of the actual engine implementation.

Chapter 7 concludes and describes future work.

Chapter 2

Background

2.1 Intensional Logic

Many natural language statements are neither necessarily true or necessarily false. The actual truth value of one of these statements depends on the context in which it is uttered. The meaning of a statement whose truth value is context dependent is its *intension*. The actual truth value of the statement in the context it is uttered in is its *extension*[5].

So the statement "I am in Australia" has the intension "If uttered in Australia then evaluate to true, otherwise evaluate to false." The extension for this statement will depend on where the person who utters the statement is actually situated.

In intensional logic the actual context in which a statement is evaluated in is explicitly controlled. This control is provided by intensional operators which allow *navigation* though possible contexts. One example of this navigational control is when a context qualifiers are added to a statement. For example the statement "what is the temperature in this room at this time?" has the context qualifiers "this room" which refers to the current physical position that the statement is made in and "now" which refers to the current temporal position that the statement is made in.

Time	5am	11am	5pm	11pm
Temperature(°)	16	24	23	18

Table 2.1: Temperatures in the K.E.G Lab during the course of the day

By keeping one of the dimensions, namely physical position, at the value "The Kernel Experimentation Group Laboratory, UNSW" and allowing the other dimension, namely time, to vary over the course of the day. The following values (see table 2.1) could be calculated[1]. The actual value of the expression changes as the context of evaluation changes in the temporal dimension.

2.2 Intensional Programming

Most computer programming is based on the imperative computational model. That is a programmer explicitly states the steps that will be carried out by a computer in order to carry out some calculation. This is fine except that the programmer actually has to figure out all the required steps in the first place.

One example of a programming paradigm that does not follow the imperative computational model is in the field of databases. In database programming the basic concept is the *query*. A query is simply a demand for a value (or values) in the database. The programmer is not concerned with how the database goes about finding those values, simply with what values are wanted.

Intensional Programming takes the database paradigm and adds the concept of *navigation*. As well as being able to query for a given value, it is also possible to change the context in which the query is made by navigating though context space. Depending on the dimensional qualifiers in the computed expression (the intension) changing the context of the query could change the value (the extension) of the query. Navigation is controlled by *intensional operators*.

[1] These temperatures are by no means guaranteed to be accurate

2.3 Intensional Engines and Education

An intensional engine is a mechanism by which programs that are based on *intensional queries* can be executed. An intensional query is similar to a database query except that in addition to specifying what is to be retrieved the context that the query is to be evaluated in, is also specified.

The standard implementation of an intensional programming system is *education*. In this system an intensional program makes a *demand* for a value calculated at some point in context space from the engine. The engine takes this demand and computes the required value. In the process of computing this value the engine may need to calculate other values, possibly at different points in context-space.

This behaviour can be seen in the following example. An intensional expression to estimate the speed of a certain particle could be:

$$speed\_estimate(now) = (position(now) - position(5 seconds ago))/5$$

In order to compute the estimated speed of the particle the engine requires two additional values. The first is the position of the particle at the current time, the second is the position of the particle five seconds ago. These values need to be demanded and these demands can be made in parallel. If the two values do not depend on each other, i.e. the position now can be calculated without knowing the position five seconds ago, then the demands can also be computed in parallel. In order to allow demanded values to be computed in parallel, intensional engines require parallel execution contexts to actually compute these values.

To prevent the same value from being computed from scratch, every time it is demanded, intensional engines include a cache of previously computed values called a *warehouse*. This enables future demands for already computed values to be satisfied immediately.

One of the problems with intensional engines is the potential for fine granularity in demands. The computation of a given demand may require the handling of hundreds or even thousands of other demands. The overhead involved in dealing with these demands can far exceed the actual processing involved in computing these demands. There are two ways to deal with this problem. The first involves increasing the granularity of the intensional system, the second involves reducing the overhead in dealing with demands so that a finer granularity can be efficiently achieved.

The Web as an Intensional System

As has been demonstrated by Plaice and Paquet[6], It is possible to consider the World Wide Web (WWW) as an intensional system. The context space for a WWW session is made up of the current *universal resource locator* (URL) and time. The standard query is to view the current page. The query result is of course the viewing of a web page. The result acquired when a view query is made by a web browser will depend on both the current URL and the time the query is made. Web browsers allow control over the current URL context by following links on web pages and by allowing the user to enter arbitrary URLs in the browser. Web browsers also allow limited navigation though the time dimension by the BACK and FORWARD operations. These allow the user to navigate through the history of pages viewed in the current session.

The concept of education can also be seen in the way a web browser displays a web page. When a demand to view a web page is made by the user. The browser retrieves the source code for the page across the network from the remote web server. In order to actually display the page however other media such as images, sounds, and possibly other web pages. The browser will open up parallel network connections in order to retrieve the required media. In order to speed up access to often accessed pages, and reduce network traffic, the browser will maintain a cache of retrieved media. When a demand is made for media already in the cache, the cached copy is used to enable the browser to display the page without having to retrieve it across the network.

2.3.1 Examples of Intensional Engines

pLucid[7]

The pLucid interpreter was one of the first intensional engines. It was an interpreter for the intensional language Lucid(see 3.1). It ran on standard UNIX platform, and was implemented as a demand-driven engine. At the time that pLucid was written the Lucid language only handled contexts with one dimension.

GLU[3]

The GLU (Granular Lucid) programming environment attempts to address the granularity issue by increasing the granularity of the intensional system. GLU uses standard ANSI C functions in concert with the Lucid intensional language. Some demands are computed by user supplied C functions. As these C functions will not generate additional demands and may require a large amount of computation to actually calculate a return value, the amount of demands in the system can be kept to a manageable level.

The GLU system is a parallel programming environment that can propagate demands between multiple hosts on a TCP/IP network. Obviously there is quite a high overhead in communicating across such a network, so this is only efficient if there is significant processing to be performed in satisfying each propagated demand.

GLU is the most current implementation of the intensional programming language Lucid (See chapter 3). As such it is a heavy influence on the design of the IE4 engine and will be referenced throughout this thesis.

Chapter 3

The Ilucid Language

3.1 Lucid

The Lucid language originated in 1974 through the work of Ashcroft and Wadge. It was originally a data-flow language where streams of data were defined in terms the first and following elements in the stream. Streams were processed in sequential order, starting at the first element and proceeding on to subsequent elements. The language became intensional when the ability to access streams in a non sequential fashion was added. This meant it was possible to navigate through the stream indexes and query the stream at an arbitrary point. This meant that streams ceased to be infinite sequential sequences of values, and became intensional expressions that are evaluated based on the current context of evaluation. It was at this point that the first Lucid implementations, Luthid and pLucid, appeared. [7]

Greater flexibility to the Lucid language came with the *Indexical Lucid* which added multi-dimensional streams to the Lucid language. No longer was data limited to simple sequences of values. Streams that varied in multiple dimensions simultaneously were now possible. This opened up multidimensional data such arrays to the Lucid language greatly increasing its possible application to scientific programming. This is essentially the version of Lucid implemented in the GLU compiler[3].

Extensions to the language have been proposed. Such as the ability to use functions as values, and the ability to use dimensions themselves as values[5].

The IE4 language will use a slightly simplified version of the multi-dimensional Indexical Lucid. This language will be referred to as *Ilucid*.

3.2 Ilucid

Ilucid is very similar to the intensional portion of the GLU language as described in the GLU Programmer's Guide[3]. The following sections especially the non-pointwise operators draw heavily from definitions in that document.

3.2.1 Types

Ilucid as implemented in the IE4 system supports long integer, double, and string types. In standard C fashion the boolean type is an abstraction of the integer type.

3.2.2 Identifiers

As Ilucid is not an imperative procedural language their is no real concept of a 'variable'. An Identifier is a named expression. Identifiers are declared with an assignment operation. The identifier is on the left and the expression that is being assigned to the identifier is on the right. Identifiers always denote a (possibly multi-dimensional) sequence of values through context space.

3.2.3 Constants

Constants in Ilucid are simply identifiers that have the same value regardless of the context in which they are evaluated. Constants can be integers or strings.

3.2.4 The `where` clause

The way programs are structured in Ilucid is with the `where` clause. The `where` clause behaves similarly to the where clause in the functional language `miranda`. It allows subexpressions to be replaced by identifiers. Thus the function

```
X = ( A + B / C) * (A - B);
```

can be rewritten as:

```
X = Y * Z
where
   Y = A + B / C;
   Z = A - B;
end
```

3.2.5 Dimensions

New dimensions in Ilucid are introduced with the `dimension` declaration. New dimensions have initial value 0.

3.2.6 Operators

Ilucid has a number of operators that are used to define computations. *Pointwise operators* compute a result at a certain point in a sequence. They require only values calculated in the current context. *Non-pointwise operators* are any operators that cause a change in the computational context. These are the intensional operators of the Ilucid language.

<table>
<tr><th colspan="2">Arithmetic Operators</th></tr>
<tr><td>addition</td><td>+</td></tr>
<tr><td>subtraction</td><td>-</td></tr>
<tr><td>multiplication</td><td>*</td></tr>
<tr><td>division</td><td>/</td></tr>
<tr><td>modulo</td><td>%</td></tr>
<tr><td>negation</td><td>~</td></tr>
<tr><th colspan="2">Relational Operators</th></tr>
<tr><td>Greater Than</td><td>></td></tr>
<tr><td>Greater Than or Equal To</td><td>>=</td></tr>
<tr><td>Equality</td><td>==</td></tr>
<tr><td>Non Equality</td><td>!=</td></tr>
<tr><td>Less Than or Equal To</td><td><=</td></tr>
<tr><td>Less Than</td><td><</td></tr>
<tr><th colspan="2">Logical Operators</th></tr>
<tr><td>And</td><td>&&</td></tr>
<tr><td>Or</td><td>||</td></tr>
</table>

Table 3.1: Arithmetic, relational and logical operators in Ilucid.

Pointwise Operators

Ilucid has the standard arithmetic, relational, and logical operators. These are shown in fig 3.1.

Non-Pointwise Operators

The simplest non pointwise operators are `first` and `next`. `first` returns a stream where every element is the first value of its operand sequence. `next` returns a sequence that is identical to the operand sequence except the first element has been removed. `first` and `next` are closely analogous to head and tail in list operations.

```
A                        = [0]  1   2   3   ...
first.time A             = [0]  0   0   0   ...
next.time A              = [1]  2   3   4   ...
next.time next.time A    = [2]  3   4   5   ...
```

Operator `fby`

The operator `fby` (followed-by) takes two sequences and returns a sequence whose first element is the first element of the first sequence and whose following elements are the second sequence.

```
A                        = [0]   1   2   3   ...
B                        = [5]   6   7   8   ...
A fby.time B             = [0]   5   6   7   ...
A fby.time next.time A   = [0]   1   2   3   ...
```

Operator `asa`

The operator `asa` (as-soon-as) accepts two sequences and returns a constant sequence whose reoccuring element is the kth value of the first sequence where k is the index of the first true (non zero) value of the second sequence.

```
P              =   [0]   0   0   0   0   1   0   1   1   ...
A              =  [42]  43  44  45  46  47  48  49  50   ...
A asa.time P   =  [47]  47  47  47  47  47  47  47  47   ...
```

Operator `wvr`

`wvr` takes two sequences and returns what is essentially the first sequence filtered by the boolean status of the second. The result is made up of those operands in the first sequence where the corresponding operand in the second sequence evaluated to true.

```
P              =  [0]   1   0   0   0   1   1   1 ...
A              =  [42] 43  44  45  46  47  48  49 ...
A wvr.time P   =  [43] 47  48  49  ...
```

Operator `upon`

The operator **upon** (upon) also takes two sequences. It returns a stream whose kth value is the pth value of the first sequence such that p of the first k values of the second sequence are true.

```
P               =  [0]   0   1   1   0   1   0   1  ...
A               =  [10] 12  14  16  18  20  22  24  ...
A upon.time P   =  [10] 10  10  12  14  14  16  16  ...
```

Operators @ and

The @ (at) operator represents the primitive intensional idea of navigation through context space. It takes two sequences and returns a sequence whose values correspond to the first sequence indexed by the value of the second sequence at each point.

The # (hash) operator allows an explicit query of the current index into a given dimension. So #.time would return the value of the time dimension in the current context.

```
A               = [1]  2  4  8  16   32  64 128  ...
B               = [1]  2  3  0   6    7   4   5  ...
A @.time B      = [2]  4  8  1  64  128  16  32  ...
#.time          = [0]  1  2  3   4    5   6   7  ...
```

`if-then-else` Operator

The `if-then-else` construct takes one boolean sequence and 2 other sequences. The returned sequence value is made up from an element of the first sequence if the boolean sequence value is true or the corresponding element from the second sequence otherwise.

```
P                       = [1]  0    0    1    0    1    0    1    ...
A                       = [42] 43   44   45   46   47   48   49   ...
B                       = [6]  5    4    3    2    1    0    -1   ...
if P then A else B fi   = [42] 5    4    45   2    47   0    49   ...
```

3.2.7 User Defined Functions

Ilucid allows for the definition of user defined functions. Functions in Ilucid are not necessarily point-wise. The parameters of the function are not values calculated at a point but entire streams. An example of this is shown below.

```
avg.d(X) = (X + next.d X) / 2;

A                 = [0]   2   4   8  16  20   ...
next.time A       = [2]   4   8  16  20  30   ...
av.time(A)        = [1]   3   6  12  18  25   ...
```

As it cannot be certain at call time in what context the sequence will need to be evaluated, parameter values are computed as needed in the function body. This "call by need" parameter evaluation essentially means that functions behave like identifiers, they are aliases for expressions. Unlike identifiers however functions cannot be defined recursively[1].

[1]This is a simplification from the standard version of Lucid. This restriction greatly simplifies the handling of function calls in the engine

Chapter 4

IE4 and Code Conversion

4.1 Interface Overview

The IE4 intensional system is intended to take intensional Ilucid code and execute it on the intensional engine running on the L4 μ-kernel. However running the Ilucid code in an interpreted manner would be far to slow and direct compilation of the intensional code is far too problematic for this thesis.

An alternative to direct compilation is the conversion of Ilucid code to C code suitable for running in the IE4 engine. IE4 requires a set of functions that given a contextual argument can compute the value of an Ilucid identifier. The IE4 system provides a number of library functions to support this client C code.

The basic structure of the IE4 interface is that every identifier in the Ilucid code is converted into a C function that can be called by the engine to calculate the identifier at an arbitrary context. IE4 provides library functions that simplify the manipulation of contexts and values for client C code. It also supplies the equivalent of a remote procedure call (RPC) interface for handling the demanding and supplying of values by these computational functions.

```
typedef enum { INT_OBJ, STRING_OBJ, DOUBLE_OBJ } obj_type_t;

typedef union {
    dword_t i;
    char *s;
    double d;
} value_t;

typedef struct {
    obj_type_t type;
    value_t value;
} object_t;
```

Figure 4.1: Structure of an IE4 object.

4.1.1 Objects and Contexts

In order to allow for future flexibility all values in IE4 are stored in objects. This also allows the engine to mostly ignore the types of the values as they are being passed around inside the generator. Currently only a limited set of operations are available on objects. They can be created, cloned, and compared for equality. The values stored in the object can be directly accessed, though not in a type independent way. The structure of an object is shown in fig 4.1.

Currently the only objects that are supported are objects containing integer, double, and string data types. This could easily be expanded to add additional types to IE4 and Ilucid.

As well as the values that are computed by the engine the most important data type is the context. The context defines in what position in context space the current query is being evaluated. Contexts in IE4 are limited to a maximum of 64 dimensions.

A context consists of a set of dimension values. These values are specified by the **rankmap** bit field. The actual values for the dimensions given in the context are stored in objects attached to the context itself. Currently only integer objects are valid dimension values.

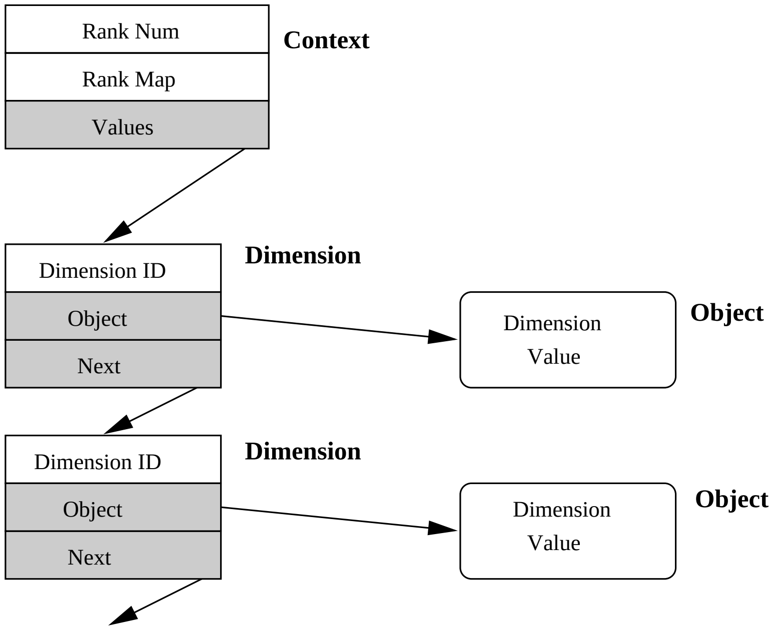

Figure 4.2: The structure of the context data type.

4.1.2 Lightweight Functions

Some identifiers have a very low computation cost. These are identifiers where the calculations required to compute the value are very simple. For these identifiers it is not efficient to spawn a remote worker, even in the low overhead L4 environment. In this case the value is calculated inside the generator itself using a *lightweight function*.

A lightweight function for an identifier takes the current values of dimensions in its rank and returns the value for the identifier in that context. Lightweight functions have to adhere to a number of restrictions.

- Lightweight functions may need the values of other identifiers to complete their computation. Any identifier needed however must also be lightweight.
- If a lightweight function does call another lightweight function to perform a subcomputation then it must use the warehouse access hooks (see 4.1.3) to update the warehouse with the subcomputation result.

```
if (not vvc_get(identifier, context, &value)) then
    value = lightweight_function(identifier, context)
    vvc_add(identifier,context,value)

if (need to make changes to value)
    copy value
    use new copy
```

Figure 4.3: Pseudo code for warehouse access in lightweight functions.

- No object or context passed to the warehouse may be modified in any way afterwards. (Heavyweight functions do not have this restriction as they terminate after passing their result to the warehouse.)

In order to comply with these restrictions most requests for subcomputations in a lightweight function will follow pattern shown in fig 4.3.

4.1.3 Function Table

The most important part of the IE4 interface is the function table. This table is loaded into the address space of the engine and the information provided in the table is used by the engine to compute demanded values.

The function table should be initialised with the appropriate values by the client `main` function. The only things that the `main` function should do is to initialise a function_table data-structure with appropriate values and then call the library function `init_function_table` (see 4.1.4).

numidents: The number of identifiers in this function table.

vvc_get: A pointer to the warehouse search function. This value is filled in by the generator at runtime. It is present so that the function can be called by lightweight functions. The arguments to the function are respectively.

```
typedef struct function_table {
    dword_t numidents;
    int (*vvc_get)(dword_t,void *,void **);
    int (*vvc_add)(dword_t,void *,void *);
    functions_t func[1];
} function_table_t;

typedef struct {
    dword_t lightweight;
    dword_t rankmap;
    dword_t heavy;
    void *(*light)(void *);
} functions_t;
```

Figure 4.4: Structures for the function table.

- The identifier being queried.
- The context of the query.
- A pointer to the location were the result of the query can be stored.

This function returns true (non-zero) if the required value was found in the warehouse.

vvc_add: A pointer to the warehouse insertion function.This value is filled in by the generator at runtime. It is present so that the function can be called by lightweight functions. The arguments to the function are respectively.

- The identifier being inserted.
- The associate context.
- The object representing the computed result.

func: The beginning of the identifier array. This is the array of function info as described below.

lightweight: A boolean flag if this is non zero then the function is lightweight and should be computed in the generator itself. Depending on this value only one of the following two fields will be valid.

rankmap: A bit-field that shows which dimensions are in the rank of this identifier.

heavy: The initial IP for a worker that computes the non-light identifier.

light: A pointer to the function that computes the lightweight identifier.

Function-0

The first function in the function table is *Function-0* this is the entry point for the intensional code. It is run by the engine automatically when the engine starts. *Function-0* is always a heavyweight function.

4.1.4 Library Interface

main() Interface

The following function can only be called by the client main() function.

```
void init_function_table(void *table)
```

This function takes a pointer to the client function table which is used to initialise IE4.

Context Functions and Intensional Operators

These functions are those context manipulating functions that are applicable to client code. They can be used by both heavyweight and lightweight functions.

```
/* context handling functions */
context_t *context_new(void);
context_t *context_clone(const context_t *context);

void add_dimension(context_t *context, unsigned d, object_t *v);

/* intensional operators */
context_t *at(context_t *current, unsigned dim, int v);
int ihash(context_t *context, unsigned dim);
```

context_new: Returns a new context with no dimensions.

context_clone: Clones a context returning the new copy.

add_dimension: Adds a dimension to a context.

at: returns a new context whose dimensions are identical to the passed context except dimension `dim` now has value `v`.

ihash: returns the value of dimension `dim` in the given context.

Heavyweight Function Interface

The following functions are available only to heavyweight functions:

`context_t *my_context(void)`
This function returns the evaluation context of the function.

`void add_demand(ident_t i, context_t *c, demand_list_t **list)`
This function adds a demand for identifier `i` evaluated at context `c` to the demand list `list`. If `list` has not been initialised then it will be initialised by this function.

`void demandAndWait(demand_list_t *list)`
This sends a message to the engine, which demands everything in the demand list `list`. The function will not return until all demands in `list` have been satisfied.

`object_t *get_result(int query, demand_list_t *list)`
This function returns the `query`th result from the demand list `list`. It is only useful after the `demandAndWait()` call for `list`.

`void supply(object_t *value)`
This function sends `value` to the engine as the computed result for the callers computation. It causes the caller to terminate.

4.2 Code Conversion

The translation from Ilucid code to these C functions that can be run on the IE4 intensional engine takes a number of steps. A parse tree is generated from the user Ilucid code. A number of operations are then carried out on this tree. Firstly user defined functions are expanded out at each call. Secondly all derived non-pointwise lucid operators are reduced to their primitive form. Thirdly rank analysis is performed on all identifiers. Finally the primitive code is converted to C code that can be compiled and linked against the IE4 user libraries.

4.2.1 Dealing with Functions

As stated in the language definition in chapter 3 all functions in Ilucid are aliases for their underlying expression. The function can be removed from the parse tree by expanding out the function at each call site. The actual call-site parameters of the function are substituted for the defined parameters of the function in the function body. Identifiers local to the function are tagged differently for each call-site. This behaviour is shown in the example below.

```
A where
  dimension d;
  A = 1 fby.d merge.d(2*A,3*A);
  merge.a(x,y) = if (xx <= yy) then xx else yy fi;
  where
    xx = x upon.a (xx <= yy);
    yy = y upon.a (yy <= xx);
  end;
end;
```

becomes

```
A where
  dimension d;
  A = 1 fby.d (if (xx_tag1 <= yy_tag1) then xx_tag1 else yy_tag1 fi;
  where
    xx_tag1 = 2*A upon.d (xx_tag1 <= yy_tag1);
    yy_tag1 = 3*A upon.d (yy_tag1 <= xx_tag1);
  end;
end;
```

This approach is a simplification that only works because functions in Ilucid have been defined to not include recursive definitions. A more powerful and flexible system has been proposed that can deal with recursive definitions and functions of greater than first order (i.e. functions that take functions as arguments) by adding two additional intensional operators to the Lucid language[6]. As this is not part of the IE4 implementation this technique will not be explained here.

4.2.2 Reduction to Primitive Form

As described in chapter 2 the foundations of intensional programming are the ideas of navigation and querying. The operators `@`,`#`, and `if-then-else` allow these concepts to be explicitly defined. As the operations defined by these operators are what all intensional operators are based around, it is also possible to reduce all of the previously defined operators to their *primitive form*.(re JOEY) This is shown below. For clarity the more complicated definitions are not reduced completely to the primitives, but are instead left in terms of other operators that could be reduced to primitive form if required.

For arbitrary dimension d.

```
first.d X     =  X @.d 0
next.d X      =  X @.d (#.d + 1)
X fby.d Y     =  if #.d == 0 then X else Y @.d (#.d - 1)
X wvr.d Y     =  X @.d T
                  where
                    T = U fby.d U @.d (T + 1)
                    U = if Y then #.d else next.d U
                 end
X asa.d Y     =  first.d (X wvr.d Y)
X upon.d Y    =  X @.d W
                  where
                    W = 0 fby.d if Y then (W+1) else W
                  end
```

4.2.3 Rank Analysis

An intensional expression is unlikely to have its extension changed by changes in the value of every possible dimension. For example the expression.

$$pressure = depth \times constant$$

depends only on the value of the dimension depth. Changing the values of other dimensions, like time for example, will never affect the computed value for pressure. The set of dimensions upon which an intensional expression is actually dependent is known as the *rank* of the expression. In Ilucid expressions are aliased by identifiers. The rank of an identifier is equal to the rank of the right hand side expression it is derived from.

Rank Analysis in the GLU compiler[1]

GLU performs rank analysis by applying a set of simple rules to all identifiers in the parse tree in a bottom up manner. These rules are applied in multiple passes of the tree until the calculated ranks do not change between iterations. The rules used by GLU are shown below. There are also additional rules used to handle functions which are not shown.

$$
\begin{aligned}
\text{Rank}[\textit{constant}] &= \emptyset \\
\text{Rank}[\#.\text{d}] &= \{\text{d}\} \\
\text{Rank}[E_1 \text{ op } E_2] &= \text{Rank}[E_1] \cup \text{Rank}[E_2] \\
\text{Rank}[\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ fi}] &= \text{Rank}[E_1] \cup \text{Rank}[E_2] \cup \text{Rank}[E_3] \\
\text{Rank}[E_1 @.\text{d } E_2] &= (\text{Rank}[E_1] \text{ - } \{\text{d}\}) \cup \text{Rank}[E_2]
\end{aligned}
$$

Rank Analysis in IE4

Rank analysis in IE4 follows the GLU model with regard to basic functions. According to Dodd[1] the main limitation of the GLU rank analysis system is the compromises used in dealing with functions. By ensuring that functions are removed from the parse tree these compromises can be avoided.

Optimisation

Rank analysis allows redundant @-expressions to be eliminated. So operations can be simplified as follows

$$E_1 @.d\ E_2 \Rightarrow E_1 \qquad \text{if d} \notin \text{Rank}[E_1]$$

This only applies if the E_2 is known to terminate. This means that it is most often used to eliminate `first` and `next` expressions on dimensions not in the rank of E_1.

4.2.4 Translation to Engine C

Once the Ilucid code is in primitive form the only operators present are point-wise operators and the intensional primitives `@`, `#` and `if-then-else`.

At this point each identifier in the Ilucid code is assigned a unique index. This number is what is used to reference the identifier in demands and warehouse accesses. Similarly each dimension is assigned an index value which is used to refer to that dimension in the context manipulation functions.

The operators `@` and `#` are translated to the library functions `at` and `ihash` respectively. The `if-then-else` operator translates naturally to the C if construct. Pointwise operators are converted to their equivalent C operator.

An simple worked example of Ilucid to Engine C translation is given in Appendix A.

Chapter 5

OS Support

5.1 The L4 μ-kernel

A kernel is that part of an operating system (OS) that executes in the processor's privileged mode. In this mode most security and protection mechanisms built into the processor are disabled. A μ-kernel is an implementation of a kernel where the functionality provided by the kernel is kept to a minimum. This approach reduces the amount of code which must be absolutely trusted because it is running without any restrictions on its behaviour.

The services that are present in a μ-kernel are those services that are impossible to provide outside of the kernel without affecting the integrity of the system as a whole. As the amount of code in the kernel is kept to a minimum, the code that is present can be optimised far more heavily than code in a more complex kernel. So those services that are provided by a μ-kernel such as L4 should be very efficient.

5.1.1 L4 Abstractions & Mechanisms

The major abstractions provided by L4 are:

- address spaces: allow the use of virtual addresses which may or may not be mapped to an arbitrary location in physical memory. User processes can use a virtual address without being concerned with where the address is actually stored on the physical machine.

- threads: are the basic execution abstraction. A thread will execute arbitrary code in an address space. Threads are extremely lightweight and cheap to create, destroy, start, and stop.[reference L4 user manual]
- inter-process communication (IPC): allows two threads to communicate even if they are in separate address spaces. All IPC in L4 is synchronous so both sender and receiver must be ready for communication before the IPC can commence. IPC in L4 has very low overhead[4].

A more detailed description of the L4 μ-kernelcan be found in the reference manual[2].

5.1.2 Application to an Intensional Engine

What makes L4 a good candidate for an intensional engine is the fact that two of the mechanisms it provides directly are needed by an efficient Intensional Engine. The first of these is lightweight execution contexts (threads). These are logically parallel execution contexts needed by an intensional engine to compute demands in parallel. The current implementations of L4 run on uniprocessor machines so threads do not provide actual parallel execution. However as L4 implementations move to multi-processor machines, actual parallel execution will be available.

The second is low cost IPC. One problem encountered by previous intensional engine implementations is that the communication overhead between the engine and the parallel execution contexts is very high. This means that, except for very computationally intensive demands, the cost of generating and propagating the demand is prohibitively expensive compared with the actual benefit of parallel execution.

5.2 iOS

The aim of this thesis is to implement an intensional engine that has to deal with as little OS overhead as possible. Given this it seems strange to introduce an additional layer between the engine and the machine. However the design simplification that

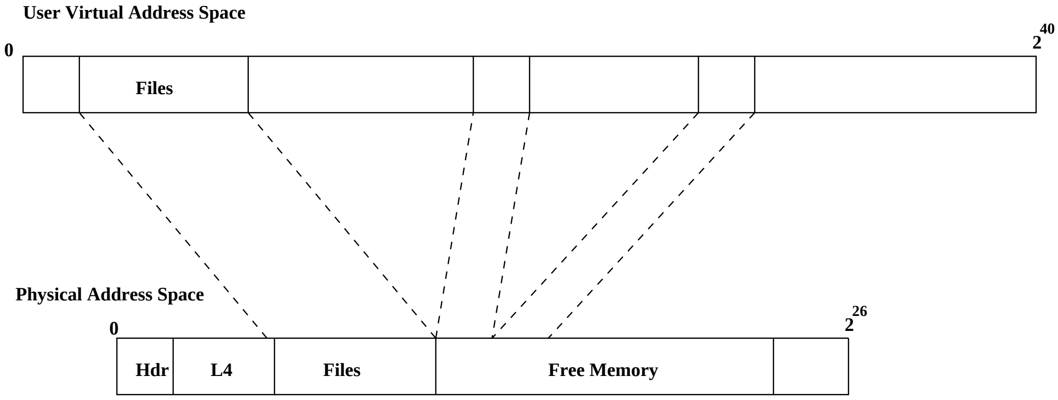

Figure 5.1: Address space provided by iOS

this allows, as well as the fact that the extra overhead incurred is very low, make the addition of an extra OS layer acceptable. This additional OS layer is referred to as iOS.

The following services are provided by iOS over L4 are:

- Large Virtual Address Space.
- Basic serial I/O.
- Loading Client Function Table.

5.2.1 Virtual Address Space

The most important service provided by iOS is a large virtual address space. The MIPS-4600 provides a 40 bit virtual address space for user tasks. The pager provided by L4 only provides mappings with a 1-1 correspondence to physical memory. The iOS pager allows the full 40 bit address space to be used by its clients (such as IE4). This allows the engine to place its data structures at constant locations in the address space without having to worry about the actual memory configuration of the machine it is being run on.

The address space provided by iOS is shown in fig 5.1. iOS direct maps all memory in the boot image. This means that any code in the boot image that uses any global variables will not be re-entrant. This is not a problem as all code in IE4 that needs to be re-entrant uses only stack variables.

The cost of providing this address space is extra computation time the first time each virtual page is touched in memory. After this initial cost however the mapping is maintained by L4 so there is no additional overhead compared with using the L4 pager.

5.2.2 Loading Client Function Table

When IE4 initialises it needs to get a table of functions from the client code (see 4.1.3). The address of this table is obtained by running the main function of the client code. iOS provides a system call to provide this address to IE4. Because the boot image is direct mapped IE4 can then directly access the function table and hence client functions.

Chapter 6

Engine Design and Implementation

6.1 Design

A demand-driven intensional engine is responsible for running intensional code which may make demands that depend on the context in which they are evaluated. The engine must calculate the result of these demands and supply the computed values to the demanding code. As calculating demanded values from scratch every time is a potentially time consuming operation, demand-driven intensional engines maintain a *warehouse* of previously computed values. This allows future demands for the same value in the same context to be satisfied easily.

6.1.1 Parallel Architecture

GLU is capable of running in two main modes. This simplest is *centralised generator architecture* (CGA). In this mode there is only one warehouse and access to the warehouse is though a *generator*. The generator is responsible for querying the warehouse. If values need to be computed this task can be offloaded to a remote worker. This system is limited as the single generator can get swamped if it has to communicate with a large number of workers. The generator also represents a

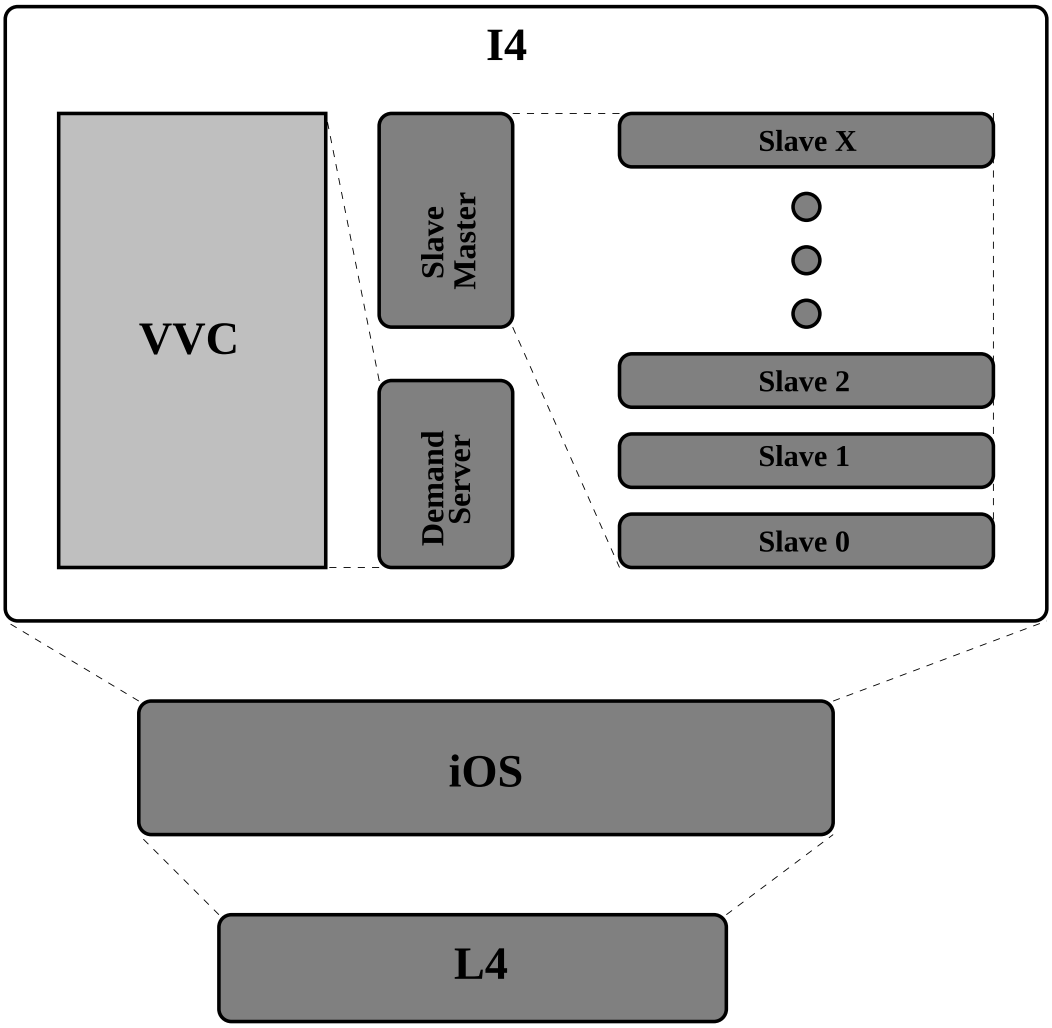

Figure 6.1: Overview of the L4 Intensional Engine

single point of failure for the entire system.

The second mode is *distributed generator architecture* (DGA). In this system there are multiple generators, and each generator has its own warehouse. Which generator is responsible for calculating a given value is determined at runtime by using some standard metric. Generators can query each other for values if needed. This architecture has the advantage that a single generator is less likely to be swamped by a large amount of communication, both demands for values and requests for computations. It is also potentially able to survive the loss of a given generator, as the other generators can detect the loss and take up the slack. It is of course a much more complicated system than CGA.

IE4 is built around the centralised generator architecture. This choice was made for a number of reason:

- IE4 does not actually run on a parallel machine so having additional processes attempting to access the VVC concurrently would only result in increased slowdown due to synchronisation controls, without any benefit.
- As the engine runs on a single machine it does not have to worry about the failure of part of the system. If the machine fails, everything fails.
- The limiting factor in this architecture is determined in part by the communication overhead between workers and the VVC [3] and in L4 this overhead is extremely low.
- Simplicity.

6.1.2 Workers and Address Spaces

In GLU calculations remote from the generator are made by *ripe function executors* or *workers*. These workers exist as distinct Unix processes either on the same machine or on a remote machine. As separate processes they do not share an address space with the generator, so all data that has to be transferred between worker and generator must be sent by messages. The separation of address spaces also protects the generator from poorly behaved workers.

Separating the address spaces of a worker from the generator would be a trivial task in L4. Simply run the generator and each worker in a separate L4 Address Space. This, however, is grossly inefficient in terms of page table overhead in the kernel. So what are the tradeoffs in moving from separated address spaces to a shared address space.

Allowing workers and generator to exist in the same address space means that the communication of demands and values can be implemented using shared memory, instead of messaging. This is especially useful if the objects being communicated are allowed to grow to arbitrary size. Shared memory is also inherently faster than messaging as there is no copying of data needed.

The disadvantage of this approach is that a poorly behaved function could trash the entire engine. However IE4 will run on a single machine. So if a worker malfunctions this will be due to errors in the code being executed. As erroneous code will in all likelihood not produce a sensible result, if it produces a result at all, the integrity of the system is compromised anyway.

Given these considerations the IE4 will run its worker functions in the same address space as the generator. This limits the number of threads that can be run to approximately 100 meaning a user level thread scheduling will be needed. This use of a single address space to run the IE4 engine is not a barrier to the eventual use of this design in a distributed system, as it is possible to maintain a single address space over a distributed system.

6.1.3 Demands

An intensional engine requires an efficient method for demands to be made to the generator. It is also important that multiple demands can be issued at the same time, as this is how intensional systems achieve their high degree of parallelism.

When a process wants another process to provide the result of a computation the usual method used is the Remote Procedure Call (RPC). The requesting process executes a procedure that handles the communication with the remote process. When the communication is finished and the result has been received the procedure returns control back to the calling process.

```
typedef struct {
    int number;
    int outstanding;
    demand_t *head;
} demand_list_t;

typedef struct demand_s {
    ident_t ident;
    context_t *context;
    object_t *result;
    struct demand_s *next;
} demand_t;
```

Figure 6.2: Structure of 'demand list' and 'demand' data types.

Given that multiple demands should be able to be made at the same time, it doesn't make sense to issue a separate RPC operation for each demand made. Instead the requester should note down every expression that it is going to need for the current computation and then issue a single RPC to the generator.

IE4 Implementation

IE4 uses this method of sending not just a single demand but multiple demands cahined together in a list format. The actual demand is sent in a register only L4 IPC message containing a pointer to the start of the demand list. The fields in these structures are shown in fig 6.2 and are described below.

number: the number of deamands in this list.

outstanding: the number of demands that have yet to be satisfied.

head: a pointer to the first demand in the list.

ident: the identifier being deamnded.

context: the context the identifier is to be computed in.

result: a null pointer that will be filled in with a pointer to the demanded value.

next: a pointer to the next element in the list.

6.1.4 Generator

The IE4 generator consists of one major data structure and two major processes[1]. The most important data structure in IE4 is of course its warehouse of already computed values. This Variable Value Cache (VVC) stores the results of all demands that the warehouse has already satisfied.

The first major process in IE4 is the *demand server*. The demand server is the gateway to the VVC. All demands for values are sent to the demand server which then queries the VVC and if necessary ensures that the required values are computed. In order to compute values in a parallel manner the demand server needs the assistance of asynchronous workers to actually compute the desired values. In IE4 these worker processes are called *slaves* and the coordination and control of these slaves is the responsibility of the *slave master*. The design and implementation of these three entities is specified in the following sections.

6.2 Variable Value Cache

The most important data structure in an intensional engine generator is the warehouse. The warehouse is where previously computed values, and their associated contextual tags are stored. This allows future demands for these value-tag pairs satisfied immediately.

As the warehouse is under constant use by the generator it is important that its operations be implemented as efficiently as possible. The nature of the warehouse is such that it only really requires two operations:

Insertion: value-tag pairs have to inserted into the warehouse for later retrieval.

Search: it has to be possible to query the warehouse for value-tag pair. This will be the most frequent operation on the warehouse. It is also important that both successful and unsuccessful searches can be carried out in minimal time.

[1] This is not a process in the normal OS usage, but a process as used in Software Engineering, i.e A flow of execution

Not that if the warehouse was going to be persistent. That is exist over a long period of time, it would also be necessary to support the *delete* operation. However as IE4 is not currently a persistent system this is not required.

6.2.1 Potential Data Structures

Two data structures that seem promising for a warehouse implementation are *binary trees* and *hash tables*.

Binary trees store data in a sorted tree structure. Insertion and deletion can be performed in $O(h)$ time, where h is the height of the tree. In a best case scenario the height of a binary tree of size n is given by $h = log_2\ n$. Unfortunately due to the nature of demands to a warehouse. It is entirely possible for value-tag pairs to be added to the structure in order. This is the worst case for a binary tree and results in the structure degenerating into a linked list. In this case a search for a value occurs in $O(n)$ time, which will be far to slow as the warehouse grows. This problem could be averted by the use of a self-balancing tree structure such as a red-black tree.

A hash table stores data in an array. The index into the array for a given element is calculated by applying a hash function to a key which is all or part of the inserted element. If two keys hash to the same index then a collision occurs. In the event of a collision additional elements with the same hash value are chained off in a linked list from the initial element. If a large number of collisions occur then a hash table also degenerates into what is essentially a linked list. The order of insertion does not affect the incidence of collisions in a hash table. Collisions are determined only by the relationship between the hash function and the actual keys stored in the table. A well chosen hash function that results in a low incidence of collision results in searches into the hash table that are near $O(1)$.

6.2.2 The IE4 Variable Value Cache

The IE4 warehouse, known as the variable value cache (VVC). Is built around a number of hash tables. Hash tables were chosen because under light to medium load the hash table performs very well. The binary-tree structure does not perform

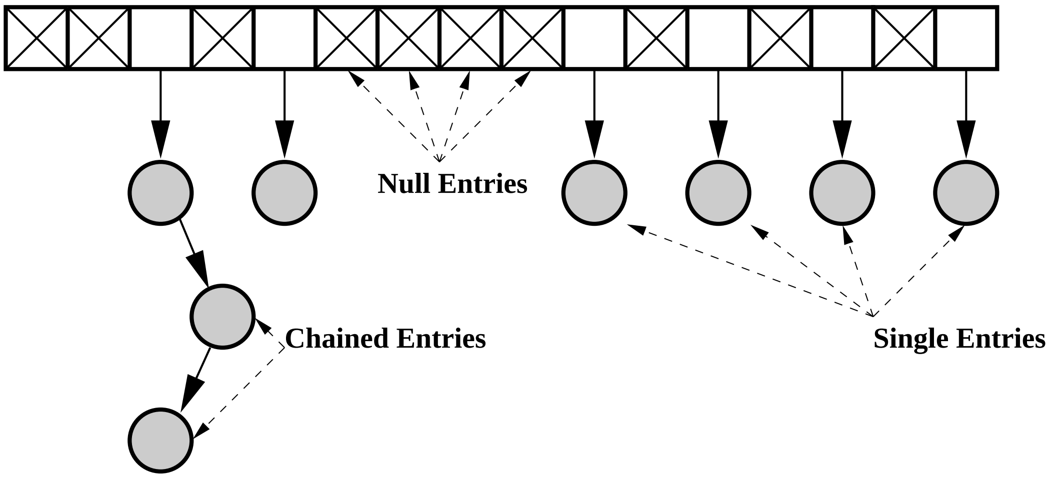

Figure 6.3: Structure of an identifier hash table in IE4.

as well at lower loads. When the VVC becomes very full then the binary tree performs very well however as IE4 is not intended as a persistent system this wasn't considered a major issue. In the future it may be advisable to combine the structures so that the chains from the hash table are actually binary trees hence gaining the advantages of both structures.

The hash tables are array of pointers with the actual entries being allocated from the dynamic memory pool.(see fig 6.3) The key used for each insertion is given by the rank of the identifier. Only contextual dimensions that are in the rank of a given identifier are used internally by the VVC for operations on that identifier. In this way queries that differ only in dimensions that are not part of the rank of the queried identifier are treated as identical. Each identifier in the VVC is allocated its own table. The overhead for this scheme is the two pages the VVC uses for the top level pointer array.

Values are inserted into the VVC by reference. Originally the VVC made a copy of every value and context it was passed. However in the case of identifiers with low computation time it was found that more time was spent copying the actual context and identifier than was spent calculating the value in the first place. Taking values by reference is considered acceptable as in last thing any remote computational function in IE4 does is to supply the value to the VVC. This means that the function that computed the value will terminate and the VVC will be left with the only reference to the supplied object.

6.3 Demand Server

The demand server is where all demands for values and supplied values are sent. It is the demand server that handles the actual access to the VVC. In initial versions of IE4 the tasks of handling demands and handling of supplied values were split between two asynchronous threads, the demand server and the value server. However it quickly became apparent that in order to maintain the integrity of the VVC and the other shared data structures in the generator synchronisation controls would be necessary. Analysis of the two threads revealed that in order to keep these shared data structures consistent it would be necessary to lock each process in one global synchronisation lock. This meant that at any one time only one of the two threads could be active, the other would be waiting on the release of the lock. So in order to simplify the implementation of the system the value server and demand server were integrated into one thread called the demand server.

6.3.1 Data Structures

In order to handle demands from multiple sources the demand server must keep track of demands currently outstanding. This information is stored in the *meta-demand-list*. Despite the name the meta-demand-list is actually a hash table. The name is merely a hold over from an earlier implementation of this data structure.

The first implementation for the meta-demand-list was simply a list of all the demand lists (see 6.1.3) provided to the demand server. This quickly proved inefficient when a value was supplied by a remote worker, as the entire list would have to be processed on each supply in order to ensure that all demands for that value were handled.

To improve performance the meta-demand-list was changed to a hash table similar in structure to the VVC. However as the meta-demand-list contains entries with dissimilar identifiers the key for the meta-demand-list includes the demand identifier as well as the demand context. As in the VVC all contexts in the meta-demand-list only include dimensions in the rank of the identifier. The relationship between entries in the meta-demand-list and the original demand list entries are shown in fig 6.5.

```
typedef struct md_hash_entry_s {
    ident_t ident;
    context_t *context;
    object_t **value_p;
    slave_info_t *owner;
    int *outstanding_p;
    struct md_hash_entry_s *chain;
} md_hash_entry_t;
```

Figure 6.4: meta-demand-list entry structure.

Each meta-demand-list entry (fig 6.4) contains the following information:

ident: The identifier being computed.

context: The context of the computation.

value_p: Where the reference to the final demanded value should be stored.

owner: The slave that made the demand.

outstanding_p: where the current outstanding count for this slave is stored.

chain: A pointer to the next entry in the chain (if present).

6.3.2 Demand Handling

When a demand list is received by the demand server the demand server systematically processes the demand list in order to ensure that all demands are eventually satisfied. A flow chart for this process is shown in fig 6.6. Each demand is processed in turn.

Rank Cull and VVC lookup

The first thing that happens to all demands entering the demand server is removal of superfluous dimensions from the evaluation context. The rank of the demanded identifier is obtained from the function table (see 4.1.3) and any dimensions present in the evaluation context that are not also in the rank of the function are removed.

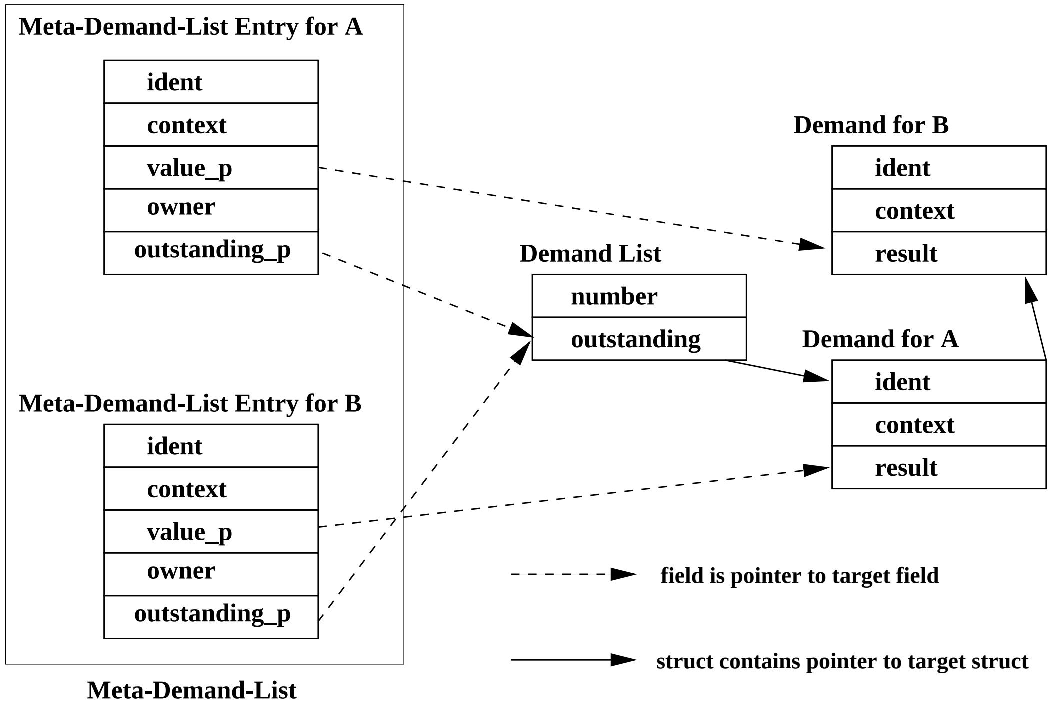

Figure 6.5: References between meta-demand-list entries and corresponding demand list entries.

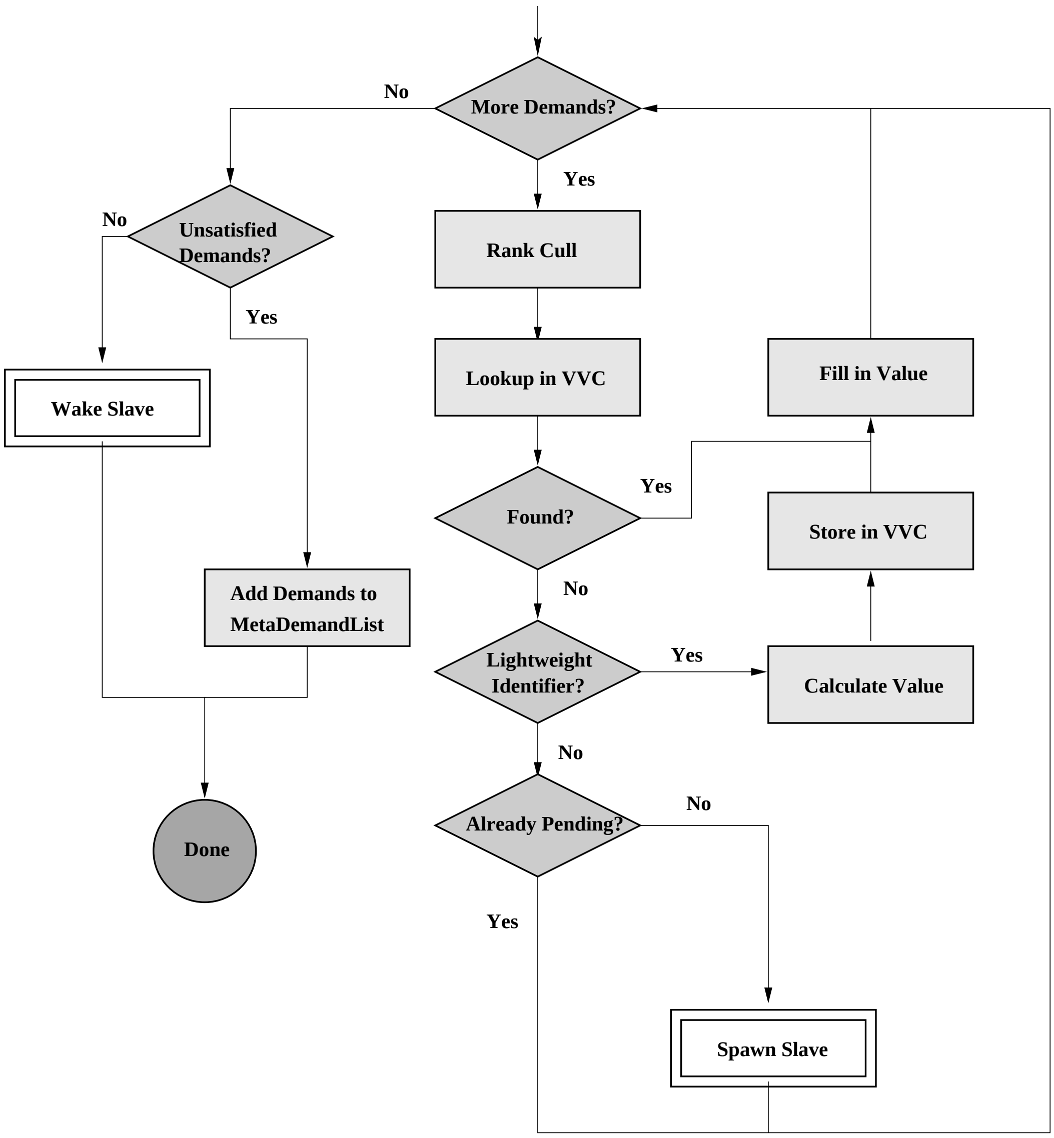

Figure 6.6: Procedure for handling demands

This forces all the internal data structures to consider demands based solely on dimensions in the rank.

After rank culling the VVC is queried for the resulting identifier-context pair. If the pair is found in the VVC the result field in the demand is filled in with a reference to the value processing for this demand ends. If the value is not present in the VVC the next step depends on weather the queried identifier has been defined as lightweight or not.

Lightweight Identifiers

Lightweight identifiers4.1.2 are identifiers where the computation is considered to be quick enough that it can be done in the generator itself and does not require the spawning of a remote slave to perform the computation. If the demanded function is lightweight the computation function is executed at this point. The computation function is responsible for adding values obtained from other lightweight functions to the VVC in the course of its execution.

Pending Check

In the case that the identifier is not lightweight the demand server checks the meta-demand-list to see if there is already an outstanding demand for this identifier-context pair. If there is a pending demand for this identifier-context pair then processing ends. If not then a spawn request is sent to the slavemaster for the demanded value.

Adding to Meta-Demand-List

After processing of all demands the demand list is checked for outstanding demands. If there are none then a wake up message for the slave is sent to the slavemaster for the demanding slave. If there are outstanding demands then all demands still outstanding are added to the meta-demand-list.

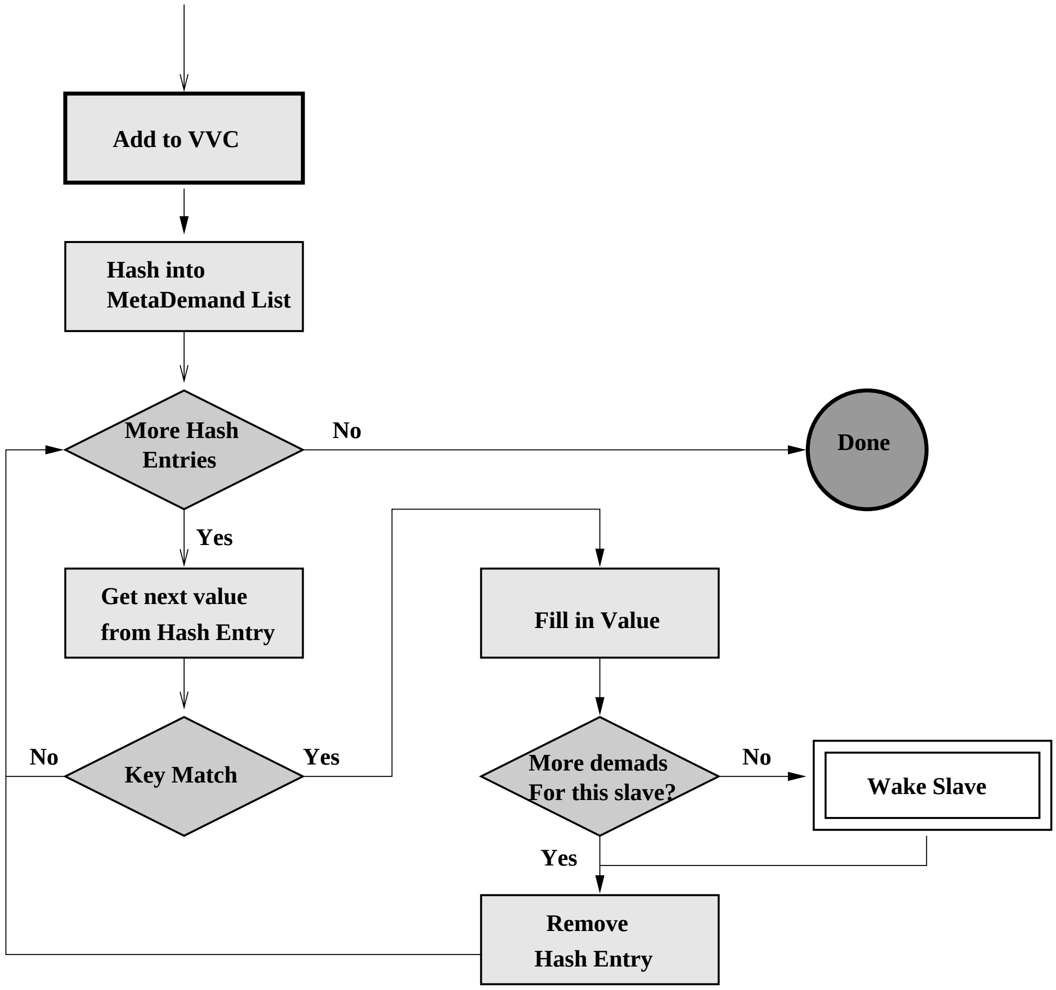

Figure 6.7: Procedure for handling supplied values

6.3.3 Handling Supplied Results

Similarly the demand server handles the arrival of supplied value in a systematic manner. This process is illustrated in fig 6.7. Throughout this process the demand server assumes that the context supplied with the value contains only dimensions in the rank of the identifier. This assumption is valid as the context for the external slave was originally supplied by the demand server. The first action of the demand server is to insert the newly acquired value into the VVC with the appropriate context tag.

Meta-Demand-List Processing

A meta-demand-list hash value is generated for the identifier-context pair. Each entry in the chain from that location is then processed in turn. If the identifier-context does not match the hash entry then processing skips to the next hash entry in the chain.

If the entry does match then the value is filled in and the number of outstanding demands for this slave is reduced by one. If there are no more demands for this slave then a wake up request for this slave is sent to the slave master. The entry is then removed from the meta-demand-list.

6.4 Slave Master

One of the most important concepts in a parallel intensional engine is the concept of a worker, who executing in parallel with the engine itself will compute a desired value.

The simpliest method to create an asynchronous execution context in L4 would be to simply create a new thread every time a worker was required. However creating a large number of threads inherently means creating a large number of tasks in which to run them.

Theoreticaly this is not much of a problem as L4 MIPS allows for the creation of 2048 tasks. This means there are potentially 2048×128 threads available to be used as workers. However in practice attempting to utilise far fewer tasks than this quickly exhausts available kernel memory with Thread Control Blocks (TCBs) and page table mappings[2].

In order to get around this problem it was decided that L4 threads should be multiplexed. So a worker will begin execution in a given thread. If the worker ever blocks, waiting on demands to be satisfied, it may be swapped out of the thread. When it is restarted it may be in a different thread to what it started in. While the worker is swapped out the thread can be of course utilised by other workers.

[2]This problem is expected to be resolved in a future version of L4 which will allow for the kernel data structures themeselves to be paged and hence grow dynamically.

```
typedef struct {
    dword_t status;
    ident_t ident;
    context_t *context;
    slave_thread_t thread;
    reg_state_t regfile;
} slave_info_t;
```

Figure 6.8: Structure of a slave_info_t(Slave Information Block).

In order to co-ordinate the spawning, swapping, and completion of these workers a dedicated server is nesecary. As the workers in IE4 are known as slaves, the obvious name for the coordinating server was the slavemaster.

6.4.1 Slaves

Workers in IE4 are called **slaves**[3]. Each slave consists of two parts, a stack used for the actual computation and information required to carry out the computation. (see fig 6.8) When a slave is said to move from the ready queue to the thread control table, it is actually a pointer to the actual slave structure that is being passed between the two structures. Each slave info contains the following information:

status: The current status of the slave.

- **free:** The slave is not currently being used to perform a computation. In this state none of the other values of the slave_info_t are valid.
- **new:** The slave has been assigned to a computation but has not actually begun execution yet.
- **used:** The slave has begun execution.

ident: The identifier being computed.

context: The context that the computation is being carried out in.

thread: Which of the threads in the engine the slave is currently executing in. If the slave is not currently in the active set of slaves this is set to an invalid value.

[3]Because I liked the name better

regfile: A register file that contains the values of the slaves registers when it gets swapped out of the active set, including the Instruction Pointer (IP) and Stack Pointer (SP).

Messaging Slaves

Theoretically the only entity slaves have to communicate with is the demand server. Either demanding values from the demand server, supplying computed values to the demand server, or obtaining demanded values from the demand server.

Communication from the slave to the demand server is not a problem as the only time the slave is able to initiate communication is when it is in the active set. So this can be simply handled by normal L4 IPC. However when the demand server wishes to respond back to demands made by a slave it has a problem. It cant be certain that the slave is still on the thread it was when it made the demand or even that the slave is on any thread at all.

Filling in demanded values is not a problem as although the execution context of the slave itself may move around, its stack and hence variables are in a constant position in the virtual address space. The demand server can simply fill in the requested values as it gets them. When it is finished filling in values the demand server still needs to signal the slave that it can resume execution.

As the demand server cannot IPC directly to the slave. It instead sends a **slave_wake** message containing the slave identifier to the slave master. The slave master then takes the appropriate action to restart the slave.

6.4.2 Slavemaster Data Structures

Thread Control Table: This is where information about the slaves currently running on l4 threads is stored.

Ready Queue: All slaves that are ready to run are stored here, they are popped off this queue in a first in first out (FIFO) basis.

Blocked Queue: This is were slaves that have been swapped out of their associated L4 thread are stored.

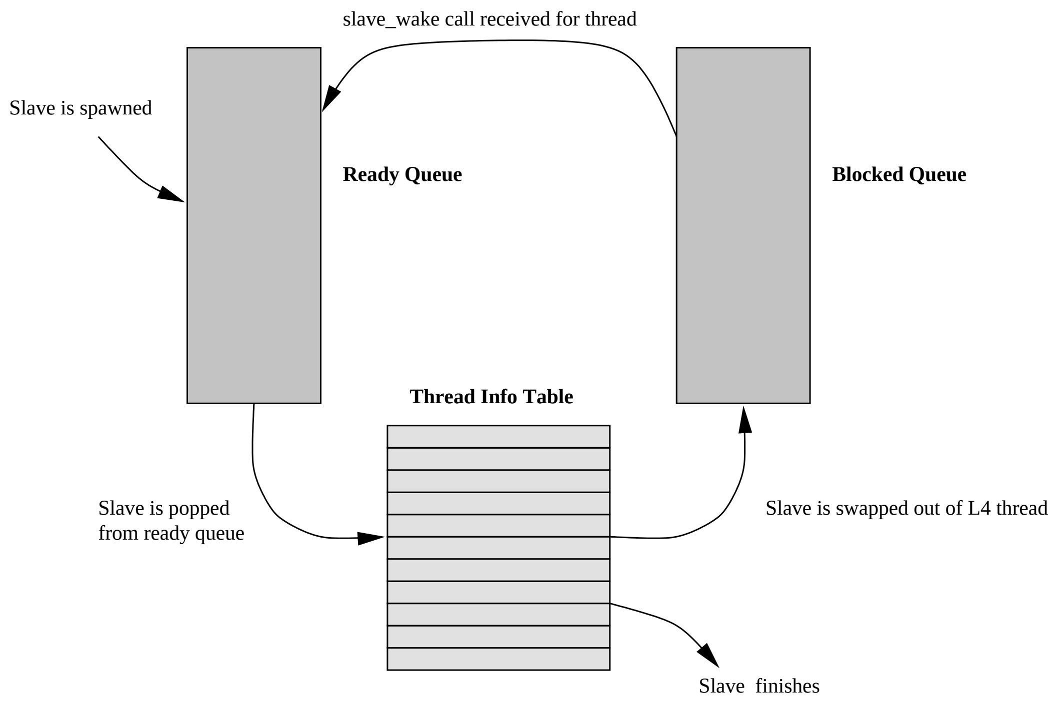

Figure 6.9: Life cycle of a slave

Thread Control Table

The thread control Table contains information for the slaves currently in the *active set*. That is slaves that are currently associated with one of the 100 L4 threads used by the slavemaster for slave execution. Slaves in the thread control table can be in one of two states. In the `running` state the slave is actively performing some computation. In the `sleeping` state the slave is waiting on the results of demands it has made to the demand server. Slaves in the sleeping state are free to be swapped out if their thread is required for the execution of another slave.

A thread control table entry may also store a pointer to another slave. This 'newslave' is used by the user level thread switching function as the target for slave switching.

When a slave finishes execution, it will nesecarily be in the active set. It notifies the slavemaster that it has finished execution via an L4 IPC message. The slavemaster in response marks that thread as `free` and attempts to pop another slave off the ready queue to make use of the now free thread.

Ready Queue

All slaves begin life here. This is a queue of all slaves that are currently awaiting execution[4]. Every time a thread becomes available either due to a slave finishing execution or a slave going into the `sleeping` state, the slave at the front of the queue is popped off and moved into the set of active threads.

The running of a slave is achieved in one of two ways. If the thread that the slave will be using is currently `free` the new slave is simply lthread_ex_reg'd into place. However if the target thread is currently in use by a `sleeping` slave, it will be nesecary to save the state of the old slave before starting the new slave.

User level thread scheduling L4 allows threads to be switched via the syscall lthread_ex_regs. However lthread_ex_regs only saves the value of the Instruction Pointer (IP) and the Stack Pointer (SP) when it switches thread contexts. This makes it is nesecary to get the thread to save its own registers and if nesecary load in a new set for itself. To achieve this to switch a task between two slaves the slavemaster calls lthread_ex_regs on the switching thread (saving the old values for SP and IP) and points the thread at code that will handle the saveing and loading of the register state.

The saving and restore of state is handled by two functions:

t_switch: carries out the saving and restoring of state for a thread. The first thing it does is to save the register state of the thread in the slaves register file. It then loads in the new register state from the new slaves register file and the lthread_ex_regs itself to the new slaves saved IP and SP values.

t_restore: loads in a register state without saving the current state, it is used when the thread being switched into is not currently in use so there is no state to save.

Obviously this is less efficient than simply allowing L4 to handle all the thread scheduling and switching operation. However as directly utilising a large number of L4 threads proved problematic, this is loss of efficiency is currently unavoidable.

[4] This is actually a lot less morbid then it sounds.

Blocked Queue

When a thread is scheduled to be switched out via t_switch it is placed in the blocked queue. It remains here until the demand server signals the slavemaster that it has fufilled all the demands made by this particular slave and that slave should be placed in the ready queue.

6.4.3 Slavemaster-Slave Interaction

`slave_spawn(identifier, evaluation_context)`
This causes the slavemaster to locate a currently unallocated slave. The slave_info for this slave is set up with the identifier being computed and the evaluation context. The initial IP of the new slave is set to the value provided in the function table. The initial SP of the slave is set to the top of the stack associated with this slave. The slave is then inserted into the ready queue for execution.

`slave_sleep \[passive\]`
When a slave in the active-set blocks waiting on demands, it sets its status flag to `sleeping`. This means that the slave is currently blocked and is free to be swapped out if it's thread is required for the execution of another slave.

`slave_wake(slave)`
This is a message from the demand server that all demands for the given slave have been satisfied and the slave can be restarted. The method of restart depends on the current status of the slave in question. If the slave is in the active set an IPC message is sent to its thread and the slave resumes execution. If the slave is currently in the blocked queue it is placed in the ready queue for execution when a thread is free.

`slave_done()`
When a slave finishes execution it sends a slave_done message to the slavemaster. This causes the slavemaster to mark the slave and its associated thread as `free`.

Chapter 7

Conclusion

IE4 proves that it is certainly possible to construct a demand driven engine on L4. It has however highlighted a number of issues in the construction of such engines.

The IE4 engine does not always gain the full benefit of the lightweight L4 threads as it was necessary to resort to user level thread switching because of memory issues in the L4 kernel. If these issues are addressed then it should be possible to move to a simpler and faster system where slaves are always in the active-set. This would eliminate the need for the blocking and ready queues and the overhead that goes with them.

Even with L4's lightweight evaluation contexts it was still necessary to introduce the concept of a lightweight thread to gain satisfactory performance from the engine. This suggests that even with a low demand overhead it is still vitally important to ensure that the granularity of parallel demands does not become to fine. This means it will be necessary to ensure that any Ilucid compiler does not naively create a large number of very lightweight identifiers.

The structure provided by IE4 should make a good base for future implementations of intensional engines, either on L4 with the intention of moving towards a multi-processing platform or to some other environment that shares the low communication overhead characteristics of L4.

7.1 Further Work

A compiler/code generator for the conversion process between Ilucid and IE4 engine C. It was hoped that this would be completed as part of this thesis but unfortunately this was not to be. This would greatly simplify the generation of new code and allow more complicated intensional code to be run on the engine. The current simplification for functions could also be removed from Ilucid.

Currently there is no metric other than 'looking' at a function to determine if a function should be declared lightweight or left as heavyweight. A cost analysis function is needed to determine this. Development of this function would require a thorough analysis of engine performance with different executing code.

The rank analysis metrics are currently very simple. In private correspondence John Plaice has suggested a more accurate method for calculating a "variable rank" for an identifier based on the actual dimensions needed to compute a value at various points in context space.

Currently the engine requires all dimensions to be fully qualified in calculating a value. That is demands can only be for a specific value. It would be interesting to be able to calculate ranges of values if a partially qualified demand is made.

Perhaps the most exciting next step is the porting of the engine to a multi-processor L4 implementation.

Bibliography

[1] Chriss Dodd. *Rank analysis in the GLU compiler, Intensional Programming I*, pages 76–82. World Scientific, Singapore, 1996.

[2] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual — MIPS R4x00.* School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 1997. UNSW-CSE-TR-9709. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[3] R. Jagannathan and C. Dodd. GLU programmer's guide. Technical report, SRI International, 1996.

[4] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for efficiency). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Cape Cod, MA, USA, May 1997. IEEE.

[5] Joey Paquet. *Intensional Scientific Programming.* PhD thesis, Département d'Informatique, Université Laval, Quebec, Canada, 1999.

[6] John Plaice and Joey Paquet. *Introduction to intensional programming, Intensional Programming I*, pages 1–14. World Scientific, Singapore, 1996.

[7] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programing Language.* Academic Press, 1985.

Appendix A

Ilucid to Engine C Example

The following pages trace through the conversion of a simple Ilucid progam to its functional equivalent in IE4 C.

Figure A.1: Initial Ilucid Code and Primative Form code

```
N @.time 30
where
  dimension time;
  N = 0 fby.time N+1;
end
```

```
N @.time 30
where
  dimension time;
  N = if (#.time == 0) then
          0
      else
          (N+1) @.time (#.time - 1)
      fi;
end
```

Figure A.2: Engine Code: #define's and function table.

```
#include <ios/syscalls.h>
#include <i4/types.h>
#include <i4/object.h>
#include <i4/intcalls.h>
#include <i4/memory.h>


void function0(void);
void function1(void);

/* BEGIN DEFINES */

#define LIGHT            1
#define HEAVY            0
#define NUMBER_FUNCTS    2

#define IDENT_0_RANKMAP  (-1)
#define IDENT_1_RANKMAP  0x1
#define ID_N               1
#define DIM_TIME           0



/* BEGIN DEFINITION OF LOCAL FUNCTION TABLE */

// ident 0 is the "begin" function and is always run initially

typedef struct {
    dword_t numidents;
    int (*vvc_get)(ident_t,context_t *,object_t **);
    int (*vvc_add)(ident_t,context_t *, object_t *);
    functions_t func[NUMBER_FUNCTS];
} local_function_table_t;

local_function_table_t lft;
```

Figure A.3: Engine Code: main function.

```
/* BEGIN MAIN */
    int main(void)
    {
        // fill in the table
        lft.numidents = NUMBER_FUNCTS;

        lft.func[0].lightweight = HEAVY;
        lft.func[0].rankmap = IDENT_N_RANKMAP;
        lft.func[0].heavy = (dword_t)function0;

        lft.func[1].lightweight = HEAVY;
        lft.func[1].rankmap = IDENT_N_RANKMAP;
        lft.func[1].heavy = (dword_t)function1;

        init_function_table((void *)&lft);

        return 1;
    }
```

Figure A.4: Engine Code: function0.

```
// entry point
void function0(void)
{
    demand_list_t *list = NULL;
    context_t *c;

    object_t *result;
    object_t *ob;

    c = my_context();

    // @.time 30
    c = at(c,DIM_TIME,30);

    add_demand(ID_N,c,&list);
    demandAndWait(list);
    ob = get_result(1,list);

    printf(``%d\n'',ob->value.i);

}
```

Figure A.5: Engine Code: function1 (Identifier N).

```
void function1(void)
{
    demand_list_t *list = NULL;
    context_t *c;

    object_t *result;
    object_t *ob;

    c = my_context();

    if (ihash(c,DIM_TIME) == 0) {
 result = new_int(0);
    } else {
 // @ (#.time - 1)
 c = at(c,DIM_TIME, ihash(c,DIM_TIME) - 1);

 // N @ (#.time - 1)
 add_demand(ID_N,c,&list);
        demandAndWait(list);
 ob = get_result(1,list);

 // N+1 @ (#.time -1)
 result = new_int(ob->value.i + 1);
    }

    supply(result);
}
```