Proceedings of the Third Workshop Computersystems

Research forum on design, realization and use of computersystems

Edited by: W. J. Withagen

Faculty of Electrical Engineering

Eindhoven University of Technology

Eindhoven, The Netherlands

May 15 1991

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Proceedings

Procedings of the third workshop computersystems: research
forum on design, realistation and use of computersystems.
Eindhoven, 15 may 1991/ ed. by Willem Jan Withagen. -
Eindhoven: University of Techonology, Faculty of Electrical
Engineering. - Fig., tab. -
Met lit. opg., reg.
ISBN 90-6144-995-2
NUGI 832
Trew.: computersystemen.

This workshop was the third in a serie of workshops organised by the research forum on design, realization, and application of **computersystems**. Currently the forum consists of researchers of four Dutch universities:

Delft University of Technology

Eindhoven University of Technology

University of Amsterdam

University of Twente

The forum also maintains a mailinglist to aid communication between the researchers. Submission of messages must be directed to:

system-network@duteca.et.tudelft.nl,

requests for inclusion or removal of this list should be directed to:

system-request@duteca.et.tudelft.nl.

Information about the forum can be requested from the university representatives:

Amsterdam	Pieter Hartel[a]	phh@ecs.soton.ac.uk
Delft	Hans Mulder	hansm@duteca.et.tudelft.nl
Eindhoven	Willem Jan Withagen	wjw@eb.ele.tue.nl
Twente	Corrie Huijs	chuijs@cs.utwente.nl

[a]Until 8/91 acting representative is Henk Muller, henkm@fwi.uva.nl

Proceedings and conference information:

Program chair	Willem Jan Withagen
Proceedings Editor	Willem Jan Withagen

Table of Contents

DALIA: A Language for the Description and Analysis of Digital Systems

Henk van der Weij
Eindhoven University of Technology
Faculty of Electrical Engineering
Digital Systems Section
P.O. box 513, 5600 MB Eindhoven, The Netherlands

Abstract

DALIA is a formal language for the description and analysis of digital systems at a behavioural as well as a structural level of abstraction. It is based on a functional style of programming, extended with annotations for indicating system structure, accessability, and signal flow direction. Program transformation rules are defined for rewriting DALIA descriptions to behaviourally equivalent alternatives. Since the same language is used for behavioural and structural descriptions, a smooth conversion between them can be made. The language is developed to be used as a formal basis for automated system design, adopting the following philosophy: First the (informal) requirements for the system to be designed are formalised in the DALIA notation. Next, the formal specification is stepwise transformed to a structural description of the system realisation (typically register transfer or gate level). The latter phase deals with formal descriptions and transformations, allowing a correctness by construction methodology.

1 Introduction

Nowadays, the technology to physically implement VLSI (Very Large Scale Integration) circuits has taken a lead on methods to correctly design them. This causes a need for formal design methods that can be automated and are able to manage the increasing complexity.

In this article, I will concentrate on a design method, and more specifically, a description language for digital systems. The design method adopted here consists of two main parts: a *specification phase* and a *realisation phase* (figure 1).

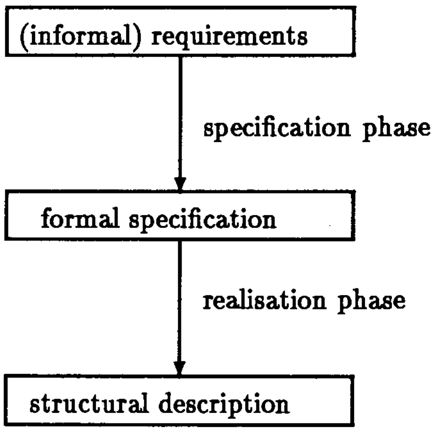

Figure 1: Design phases

Specification phase

In the *specification phase*, the informal requirements for a system to be designed are translated to a formal specification. This formal specification completely defines the system to be designed[1]. It is important to note that the specification phase can not (and will never be) fully automated. It can be computer-supported though, but the starting point will always be an informal idea.

Realisation phase

Once a formal description of the system to be designed has been obtained, its realisation can commence. The essence of the design methodology proposed here is, that this phase will be *completely formal*, and thus provable correct. The being formal of this phase does not induce that no human input is required; it only states that the actual transformation of the formal specification to a realisation is performed mechanically. The designer can 'guide' the process, and this will even be a necessity in the case of complex designs (which are the ones we want to deal with).

This translation of a 'high-level' description to a hardware structure is called *high-level synthesis* [5, 12].

1.1 Digital system description

When adopting a *correctness by construction* design methodology, a well defined description language has to be chosen for the representation of digital systems. System descriptions in this language must be suited for algorithmic manipulation to enable automation of the design path, especially the realisation phase.

In this article, a language will be introduced for the representation of digital systems under design. For this language, a number of general requirements can be stated: it must be

- Implementation independent
- Hierarchical
- General
- Formal
- Automatically processable

Currently used languages do not fulfil all of these requirements. The hardware description standard VHDL [2] for example, which is directed towards practical system design, is really useful for describing and simulating digital systems, but hardly analysable in a formal way. This is due to the large number of constructs, and the fact that it is an imperative language. Apparently, a more formal basis is needed to fulfil the requirements stated above.

A more formal approach to the modelling of digital systems is the use of formalisms for the description of communicating processes like CCS [13] and CSP [8]. These languages however are not really suited for description of data processing at a logic level. Extensions have been made to make these formalisms more practical, like LOTOS [1, 9] which is an extension of CCS. The basic language however remains *action oriented*, which makes it difficult to integrate data structures, while maintaining analysability.

[1]Theoretically, this means that the initial informal requirements can be forgotten. They do however provide non-functional, but useable information like design history and documentation

Higher-order logic is another approach to digital system description [7]. In higher-order logic, behaviour of a digital system is defined by predicates on its external lines. A circuit structure can be represented by the conjunction of predicates of the sub-circuits. Abstraction of the internal structure is accomplished by existential quantification over the internal lines [14]. Though the principle is very simple, a large class of systems can be described. A disadvantage is, that expressions tend to expand to large proportions when dealing with complex designs. Furthermore, it makes use of a discrete time model, whereas it would be desirable to abstract from time intervals (delay insensitive specification).

Another direction of research is toward description of digital systems using functional languages. The DALIA language also belongs to this class. Functional languages are easy to analyse, but not directly suited for structural description (like description of hardware architectures). In the next section, a closer look will be taken on the application of functional languages in hardware description.

2 Functional languages and hardware description

In a *functional language*, programs are described by means of functions. This essentially differs from the more common *imperative* approach, as used in languages like PASCAL and C. The imperative way of programming is induced by the one-word-at-a-time *von Neumann* concept, rather than by programmer demands [3]. In software development, the use of an imperative language can be somehow justified because of its easy implementation on conventional architectures. For the description of hardware however, imperative programming unacceptably limits the flexibility of the language.

Functional languages offer some advantages with respect to imperative languages. They are better analisable, and inherently capable of describing concurrency (arguments of a function can be evaluated in parallel). These are very desirable properties for a hardware description language.

As mentioned before, functional languages do not express structure which is a problem in case of translating a specification to a hardware architecture. This problem can be solved by changing the interpretation of function applications. For example, in [10, 15], tail recursion is interpreted as iteration, and infinite lists as signals. This solution however does not deal with another problem: the modelling of *bidirectional information flow*, a common concept in hardware designs. If this is to be handled in a formal way, other bases than pure functions have to be used like [4].

Another, more pragmatic approach is to add constructs for indicating structure like in SBL [6]. This is also the approach adopted for the design of DALIA, since it offers a basis for practical integration of the language in a hardware environment.

3 An informal introduction to DALIA

DALIA was designed by considering the most elementary characteristics of digital systems in general. The constructs of the language have been kept simple, in order to ease application of formal operations on system descriptions.

DALIA is able to describe *system behaviour* (formal specification) as well as *system structure* (architecture description). This makes it possible to use a single language as a description framework during the system design, avoiding conversion between different languages.

In DALIA, digital systems are represented by interacting concurrent *processes*. A process is an abstract information processing entity that performs part of the function of the total system. Processes may contain subprocesses, so a hierarchical system structure can be used. Furthermore, operations called *dependencies* have been defined for modelling logic operations.

3.1 Processes

Basically, a DALIA system description consists of a set of rules, defining the allowable transformations of the system. As a simple example, consider the following set of rules:

```
def (a OR 0) → a
def (0 OR b) → b
def (1 OR 1) → 1
def (NOT 0) → 1
def (NOT 1) → 0
def (a NOR b) → (NOT (a OR b))
```

These rules define the operations (functions) OR, NOT, and NOR. Expressions are evaluated by transforming them according to the given rules. An example of an evaluation is

```
(1 NOR 0) → (NOT (1 OR 0)) → (NOT 1) → 0
```

By adopting appropriate evaluation conventions, it is also possible to describe hardware-like structures. For example, an OR gate can be described as

```
def (OR_GATE a b x)
  → (OR_GATE a b (a NOR b))
```

Here a system is defined by a recursive definition. The system itself (NOR_GATE) is just a parameter, with the property that it 'remains unchanged at the same position' when applying any of the rules. This is exactly the property of a hardware resource: it is an unchangeable part of the system.

To explicitly state that parts of the system are unchangeable, the *process concept* is introduced. A process also permits local variables and data transformation rules, thus hiding irrelevant information from the outside world. A system description in DALIA consists of a set of *process definitions*

system ::=
  { *proc_def* }

In DALIA, processes are the means of structuring system descriptions. A process consists of three main parts:

- Interface
- Internal variables
- Behaviour description

Communication with a process takes place via the *interface*. The other parts of the process are invisible from the outside, hiding irrelevant details.

The *internal variables* determine the state of the process. The values of these variables can be changed by interaction of the process with its environment.

The description of the process' function is contained in the *behaviour definition*. Behaviour is defined by *data dependencies* between the variables of the process (interface and internal variables).

A variable will retain its value until it is overwritten, and reading variables is non-destructive. This provides a memory function, which reflects the register concept as known from hardware implementations.

As an example, the OR_GATE is described by the following process:

```
def (OR_GATE a b x) =
[ { a ∈ ?bit, b ∈ ?bit, x ∈ !bit }
  x = (a OR b)
]
```

or in expanded form:

```
def (OR_GATE a b x) =
[ { a ∈ ?bit, b ∈ ?bit, x ∈ !bit }
  case (a b)
    ∈ (bit 0) → x=a
    ∈ (0 bit) → x=b
    ∈ (1 1) → x=1
  endcase
]
```

Some additional notations have been introduced here, like assertions (enclosed by curly brackets), indicating type and direction of the interface variables, and a case construct. These will be discussed later.

Another important concept is the *subprocess*. The use of subprocesses is illustrated by the following process description of a NOR gate:

```
def (NOR_GATE a b x) =
[ var or_g = (OR_GATE bit bit bit);
  { a ∈ ?bit, b ∈ ?bit, x ∈ !bit }
  or_g.a=a | or_g.b=b | x=(NOT or_g.x)
]
```

or using an additional subcomponent NOT_GATE with its obvious meaning:

```
def (NOR_GATE a b x) =
[ var i,
    or_g = (OR_GATE bit bit bit),
    not_g = (NOT_GATE bit bit);
  { a ∈ ?bit, b ∈ ?bit, x ∈ !bit }
  or_g.a=a | or_g.b=b | i=or_g.x |
  not_g.a=i | x=not_g.x ]
```

Here subcomponents are referred to by variables, and the connection of components is defined by a list of assignments. The vertical bars stand for asynchronous composition (see section 3.3.3).

3.2 Values

The values used in the language are symbolic. All values are either *symbols* or *symbol trees.* A symbol is denoted by an identifier, which will by convention consist of only upper-case characters or non-letter symbols. A symbol tree is a structure that contains a number of *branches*, that are in turn symbolic values. A tree is denoted by indicating its structure using round brackets. Some examples of symbolic values are

SYM	(symbol)
1	(symbol)
(A (B C) D)	(tree)

Symbols do not have a predefined 'meaning'. For example symbol 1 is just another identifier for a symbol. An interpretation can be attached to a symbol only when operations on it are defined.

In DALIA, *sets* of symbolic values play an important role. They are used for specifying guards in conditional dependencies and for putting constraints on variables (typing). Sets will be denoted using normal set operators like $\cup$, $\cap$, and $\setminus$. In the notation adopted here, no distinction will be made between symbols and sets containing only one symbol, to ease notation. For example, $A \cup B \cup C$ denotes the set containing symbols A, B, and C.

3.3 Data dependencies

The behaviour of a process is defined by its *dependencies.* Dependencies define how variables are related to each other. A process can be 'executed' by concurrent evaluation of its data dependencies.

Basically, there are two kinds of dependencies: *assignment* and *conditional dependency*, which can be combined using the *asynchronous* and *synchronous* composition operators.

3.3.1 Assignment

The *assignment* attaches the value of an expression to a variable. It acts much like the normal assignment in imperative languages, but its interpretation differs because dependencies operate concurrently; it denotes how the current value of a variable is related to the current value of other variables in the system. The interpretation will become more clear in sections 3.3.3 and 3.3.4, where composition of dependencies is explained. An example of an assignment is

x = (a b)

which defines variable x to be equal to the combination of variables a and b.

3.3.2 Conditional dependency

Conditional dependencies can be used for the definition of logic operations on symbolic data. In a conditional dependency, (part of) the state of a process and its interface is compared to a set of symbolic values, describing a set of states. Only those subdependencies in the conditional statement that are attached to a set containing the current value of the selector expression will be activated. For example,

```
case (a b)
  ∈ (0 0) → x=1
  ∈ (1 1) → x=0
endcase
```

denotes that the value of x will be 1 if both a and b are zero, and 0 if a and b are both equal to 1. Otherwise, this dependency does not affect x. Variables unaffected by dependencies will retain their current value.

3.3.3 Asynchronous composition

With *asynchronous composition*, dependencies can be combined without specifying synchronisation between them. Evaluation of asynchronously combined dependencies is assumed to be 'fair': no statement is made about evaluation order of the dependencies or the time it takes to evaluate them, but *eventually all dependencies will be evaluated.* This assumption excludes the occurrence of *livelock.*

Though asynchronously combined dependencies are not time-related, they are allowed to share variables through which communication can take place. It is even possible to synchronise dependencies by using shared variables, but this is often circumstantial.

An example of asynchronous composition is the following:

```
y=(u v) | u=x | v=x | x=E
```

Here u, v, x, and y denote variables, and E denotes a symbol. The corresponding dependency graph is shown in figure 2. Independently of which evaluation order is applied, the result of the evaluation will be the same: x, u, v, and y will be equal to E, E, E, and (E E) respectively.

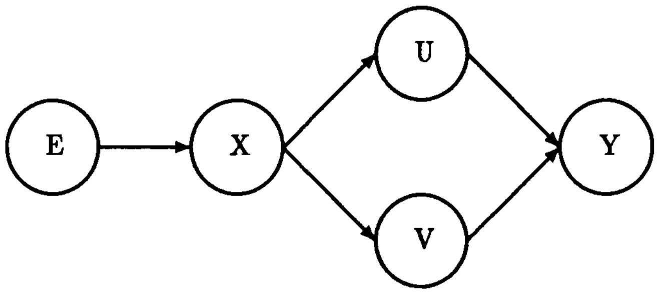

Figure 2: Dependency graph

3.3.4 Synchronous composition

For the synchronisation of dependencies, *synchronous composition* should be used. In an operational terminology: when evaluating two synchronously combined dependencies, first the result of evaluating both sides will be calculated, the combined result will be instantaneously applied on the state space. A synchronous combination can be compared to a *transition* in Petri nets [16]. It is used for introducing synchronisation points in the data-flow graph.

An example of synchronous composition is

```
x=y , y=x
```

Evaluation of this dependency will cause x and y to be 'swapped'. A dependency like this is only useful when placed within a conditional dependency, since otherwise x and y would be swapped infinitely which is unlikely to be desired. An example of such an embedded dependency is

```
case (x y)
  ∈ (0 1) → x=y , y=x
endcase
```

3.4 Assertions

For analysing descriptions, *assertions* can be inserted at various places in the process definitions. An assertion contains predicates on variables. By checking their consistency, processes can be searched for errors. Also, assertions provide information that can be used to transform processes (see section 4).

An example of an asserted dependency is

```
{ x ∈ A ∪ B }
case x
  ∈ B ∪ C → y=x
endcase
```

4 Process transformations

A *process transformation* is a function that transforms a process description to a semantically equal alternative description [11]. To give an indication of the principle, consider the asserted conditional dependency of section 3.4. The assertion states that x is a member of set A ∪ B. The condition within the case dependency specifies the case that x is a member of set B ∪ C. From this, it can be derived that in this specific case x can never be equal to A nor to C. This means that C can be removed from the set specified in the case dependency.

```
{ x ∈ A ∪ B }
case x
  ∈ B → { x ∈ B } y=x
endcase
```

which in turn is semantically equal to

```
{ x ∈ A ∪ B }
case x
  ∈ B → y=B
endcase
```

The result of these process transformations is a 'more simple' process, performing exactly the same function as the original one. This example illustrates just one of the applications of process transformations.

The aim is to formalise transformations like these, in order to automate the manipulation of process descriptions. For this, a formal definition of process transformations will be given:

Definition 1 *A process transformation is a function $t : L \rightarrow L$ with $\forall_{l \in L}(C(t(l)) = C(l))$. Here L denotes the set of all possible process descriptions and $C : L \rightarrow S$ a function that maps process descriptions on their semantics. S denotes the semantic domain of the description language.*

In the definition above, S is not defined formally since this is out of the scope of this article.

From the definition, it follows that process transformations map descriptions in the language on other descriptions under invariance of behaviour. They can be used to manipulate descriptions in a provably correct way. Process manipulations can be used to correctly optimise systems, transform specifications to architecture descriptions, or prove equivalence between systems. When automating transformations, the risk of introducing errors will be decreased, since all formal steps are correct by construction.

4.1 Primitive transformations

For transforming processes, definition 1 can be applied. It states that a process may be transformed to another process if their semantics are the same. This is however not a suitable approach for automating process manipulations. The problem is, that first a transformation has to be proposed, and only afterwards it can be proven correct. Clearly, such a trial-and-error approach is very inefficient.

So, in attempt to give 'direction' to process transformations, a set of *primitive process transformations* will be derived from definition 1 and the semantics definition of the language. Complex transformations can be performed by repeatedly applying primitive transformations.

A primitive process transformation rule will be denoted in terms of an *attribute grammar* for the language. An attribute grammar [18] is a grammar definition, with attributes attached to its elements (terminals or non-terminals). Attributes describe properties of these elements that can not (or not easily) be described in terms of syntax. For example, a variable identifier may have an attribute attached to it, denoting its type. The value of the attributes can be derived from other attributes in the parse-tree of a program. Relations between attributes are defined by *attribution rules* attached to the syntax-definition rules.

A complete attribute grammar of the language will not be given here. Only attributes occurring in transformation rules will be explained. Transformation rules will be denoted as follows:

$struct_1 \leftrightarrow struct_2$
$\{$ *condition* $\}$

Here $struct_1$ and $struct_2$ denote syntax expressions, and *condition* denotes a statement about the syntax elements and their attributes that has to hold for the equality to be true.

To achieve full processability of descriptions, the primitive rules must enable the conversion of any description to any of its semantic equivalents. This property is called *completeness* [17].

Definition 2 *A set of process transformations T is complete if and only if*
$\forall_{l,m \in L}((C(l) = C(m)) \Rightarrow \exists_{t \in T^*}(t(l) = m))$

Here T^* denotes the collection of functions that can be constructed by composing primitive transformations (i.e. any sequence $t_0 \circ t_1 \circ \cdots \circ t_n$ with t_i a primitive transformation).

It is not self evident that such a set of primitive transformations exists, but an attempt has been made to construct a powerful set, able to accomplish many process transformations. Transformations that can not be performed will have to take place using definition 1: by proposing a new process and proving it equivalent using the formal semantics definition of the language.

4.2 A set of primitive transformations

In the scope of this article, it is not possible to show all primitive transformations defined yet. Some important ones will be highlighted next.

4.2.1 Commutativity

These rather obvious transformations can be described as follows:

$dep_1 \mid dep_2 \leftrightarrow dep_2 \mid dep_1$

$dep_1 \,,\, dep_2 \leftrightarrow dep_2 \,,\, dep_1$

That these transformations are valid is clear, since the compositions are in parallel, so no evaluation sequence is specified.

4.2.2 Synchronisation

This transformation can be used to introduce or remove synchronisation between dependencies:

$$\begin{array}{l} dep_1 \mid dep_2 \leftrightarrow dep_1 \,,\, dep_2 \\ \{\ dep_1.uses \cap dep_2.affects = \emptyset \\ \wedge\ dep_2.uses \cap dep_1.affects = \emptyset \\ \} \end{array}$$

This definition states that synchronisation may be removed or inserted if the dependencies involved do not make use of variables the other one may change. Here attribute ***uses*** contains the variable identifiers referred to in the dependency, and *affects* contains the identifiers of variables that are affected by evaluating the dependency.

An example of a derivation using this transformation is the following:

```
x=z , y=E , z=F        ↔   (commutativity)
x=z , z=F , y=E        ↔   (synchronisation)
[ x=z , z=F ] | y=E
```

4.2.3 Condition calculus

Several rules are defined that deal with conditional dependencies. Some important ones will be shown here.

One of the most important rules deals with the propagation of conditions from the context into conditional dependencies:

$$\boxed{\begin{array}{l} \{\ ref \in S\ \} \\ \textbf{case}\ ref \\ \quad \in set \\ \quad \rightarrow dep \\ \quad rest \\ \textbf{endcase} \end{array}} \leftrightarrow \boxed{\begin{array}{l} \{\ ref \in S\ \} \\ \textbf{case}\ ref \\ \quad \in set \cap S \\ \quad \rightarrow dep \\ \quad rest \\ \textbf{endcase} \end{array}}$$

The next rule states that dependencies can be removed if they will never be active (ε denotes the empty dependency):

$$\boxed{\begin{array}{l} \{\ state \in \emptyset\} \\ dep \end{array}} \leftrightarrow \boxed{\begin{array}{l} \{\ state \in \emptyset\} \\ \varepsilon \end{array}}$$

Here *state* denotes the collection of all variables within the scope of the process. ε denotes the *empty dependency*.

These two rules can also be denoted using an attribute *cond* attached to each dependency. This attribute specifies the states in which the dependency may be activated. The notation using the assertion has been used since it is better readable.

The next rules deal with the removal of parts of the case dependency:

$$\boxed{\begin{array}{l}\textbf{case } \mathit{ref}\\ \quad \in \mathit{set} \rightarrow \varepsilon\\ \quad \mathit{rest}\\ \textbf{endcase}\end{array}} \leftrightarrow \boxed{\begin{array}{l}\textbf{case } \mathit{ref}\\ \mathit{rest}\\ \textbf{endcase}\end{array}}$$

$$\boxed{\begin{array}{l}\textbf{case } \mathit{ref}\\ \textbf{endcase}\end{array}} \leftrightarrow \boxed{\varepsilon}$$

A rule for creating mutually exclusive cases is the following:

$$\boxed{\begin{array}{l}\textbf{case } \mathit{ref}\\ \quad \in \mathit{set}_1\\ \quad \rightarrow \mathit{dep}_1\\ \quad \in \mathit{set}_2\\ \quad \rightarrow \mathit{dep}_2\\ \quad \mathit{rest}\\ \textbf{endcase}\end{array}} \leftrightarrow \boxed{\begin{array}{l}\textbf{case } \mathit{ref}\\ \quad \in \mathit{set}_1 \setminus \mathit{set}_2\\ \quad \rightarrow \mathit{dep}_1\\ \quad \in \mathit{set}_2 \setminus \mathit{set}_1\\ \quad \rightarrow \mathit{dep}_2\\ \quad \in \mathit{set}_2 \cap \mathit{set}_1\\ \quad \rightarrow \mathit{dep}_1 \mid \mathit{dep}_2\\ \quad \mathit{rest}\\ \textbf{endcase}\end{array}}$$

4.2.4 Other transformations

It is not possible to present all rules defined yet. Other rules take care of operations like

- Removing assignments to unused variables
- Removing declarations of unused variables
- Substitution of defines
- Expansion of subprocesses
- Substitution of variables

In the next section, a sample system description will be transformed, illustrating the use of process transformation.

4.3 An example

As a simple example to illustrate the use of process transformation, the design of a 3-bit wide incrementor will be shown (figure 3).
First, the top level of the system will be defined:

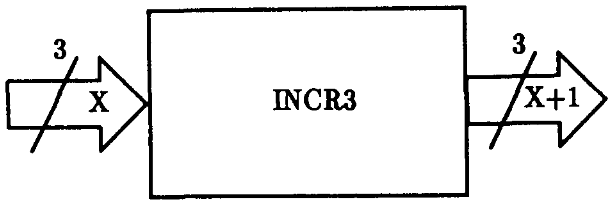

Figure 3: Incrementor

```
def (INC3 in out)=
[ { in ∈ int3, out ∈ int3 }
  out = (INC in)
]
```

The function of the incrementor has been specified by using a universal increment function defined on integer values. The data types (actually sets) used here and in the rest of this example are defined as

```
def int = ZERO ∪ (int bit)
def ZERO = (ZERO 0)
def bit = 0 ∪ 1
def int3 = (((ZERO bit) bit) bit)
```

The values modelled by elements of these sets are defined as follows:

$val((r\ b)) = 2val(r) + bitval(b)$
$val(\mathsf{ZERO}) = 0$
$bitval(0) = 0$
$bitval(1) = 1$

The definition of ZERO is unusual, since it can be rewritten only by using ZERO again. This is however no problem, because its value is well determined (0). When evaluating dependencies that make use of ZERO, it will be evaluated only if required.

Function INC can be defined as follows:

```
 def (INC x) =
 [ { x ∈ int }
   case x
     ∈ (t:int 0) → (t 1)
     ∈ (t:int 1) → ((INC t) 0)
   endcase
 ]
```

With this last definition, the system has been completed. Expansion can now start by substituting the INC function in INC3. By evaluating the initial assertions (in ∈ int3, out ∈ int3), consistency of the system can be checked. After some transformations, the result is

```
def (INC3 in out) =
[ { in ∈ int3, out ∈ int3 }
  case in
    ∈ (t:int 0) → out = (t 1)
    ∈ (t:int 1) → { t ∈ ((ZERO bit) bit) }
      out = ((INC t) 0)
  endcase
]
```

By successively substituting INC and expanding the **case** statements, eventually the following dependency (in some form) can be derived:

```
case in
  ∈ (((ZERO 1) 1) 1) →
    out=((((ZERO 1) 0) 0) 0)
endcase
```

This causes an inconsistency in the assertions since it was stated that out ∈ int3, and

```
((((ZERO 1) 0) 0) 0) ∩int3                        =
((((ZERO 1) 0) 0) 0) ∩(((ZERO bit) bit) bit)      =
((([(ZERO 1) ∩ZERO] 0) 0) 0)                      =
((([(ZERO 1) ∩(ZERO 0)] 0) 0) 0)                  =
((((ZERO [1 ∩0]) 0) 0) 0)                         =
((((ZERO ∅) 0) 0) 0)                              =
∅
```

This result indicates that an error has been made in the specification. This is indeed the fact, since the increment of 7 is 8, requiring 4 bits in binary representation. By reconsidering the initial requirements, it can be decided to 'throw away' the most significant bit or define a four bit wide output in the system specification.

5 Architecture mapping

The eventual purpose of the description and analysis language is the actual realisation of digital systems. A realisation may be a software program, but the accent of the description language is on hardware synthesis.

For translating a system description to a hardware architecture, the processes in the description have to be mapped on *resources*. These resources may be gate-level components or more complex building blocks like ALU's, memories, or even microprocessors. Within the language, it is possible to create a model of an architecture, by modelling each resource as a process and specify the connections between these resource processes by assignments.

The processes occurring in a hardware description are bound to some restrictions that are inherent to hardware implementation. For example, data types must be finite and no recursion or process creation is allowed. If the original description does contain these 'unimplementable' constructs, they have to be converted. This can be done by evaluating the constraints, that give a definition of what the process is to expect as its input. By eliminating redundant parts of constrained processes (like non-occurring data values or recursion), a finite structure may be obtained. If not, the system is either unimplementable, or the wrong decisions have been made during its analysis, leading the transformation to a dead point. This points out the necessity of human intervention during the design.

An architecture description of the incrementor example of section 4 is

```
def (INC_REAL in2 in1 in0 out2 out1 out0) =
[ { in2 ∈ ?bit, in1 ∈ ?bit, in0 ∈ ?bit,
    out2 ∈ !bit, out1 ∈ !bit, out0 ∈ !bit }
  var i=(NOT_GATE bit bit),
    a=(AND_GATE bit bit bit),
    x1,x2=(XOR_GATE bit bit bit);
  var n;
  i.in=in0 | out0=i.out |
  a.in1=in0 | a.in2=in1 | n=a.out |
  x1.in1=in0 | x1.in2=in1 | out1=x1.out |
  x2.in1=n | x2.in2=in2 | out2=x2.out
]
```

The interface has been slightly changed for reasons of readability. Figure 4 depicts the described architecture.

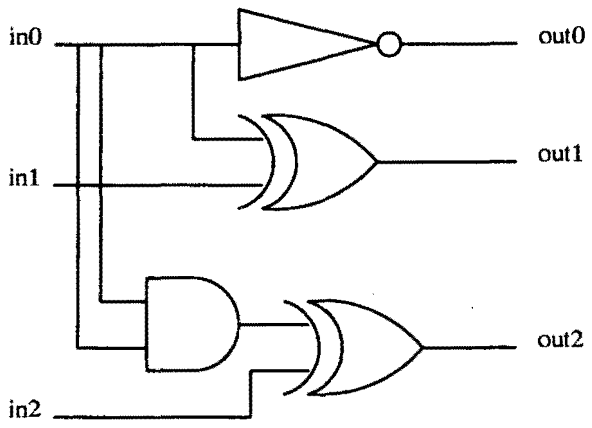

Figure 4: Increment realisation

6 Conclusions and further research

The language proposed here seems suited for the formal processing of digital system descriptions, which makes it possible to use it as a basic representation method for an automated design system. Though for complex systems it is difficult, if not impossible, to fully automate hardware synthesis from a given specification, the language may prove to be useful in offering a formal design framework. This framework is capable of guaranteeing correctness of all design steps, whether taken autonomously by the system or by the designer.

Further research will concentrate on automatic decision making. An 'expert system' layer controlling the language framework will store information from previous designs and apply heuristic searching techniques to take standard design steps or generate design proposals for the designer. It is not likely that fully automatic system design is possible in the near future, but new methods will certainly ease the designers' task.

References

[1] LOTOS: Language for the temporal ordening specification of observational behaviour, international standard ISO/IS 9074, 1987.

[2] IEEE standard VHDL language reference manual, 1988.

[3] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. In *Communications of the ACM*, pages 613–641, August 1978.

[4] Raymond T. Boute. The beta calculus: Scoping and substitution in formal descriptions of systems with bidirectional information flow. Technical Report 92, University of Nijmegen, 1986.

[5] Raul Camposano and Wolfgang Rosenstiel. Synthesising circuits from behavioral descriptions. In *Transactions on Computer Aided Design*, pages 171–180. IEEE, February 1989.

[6] Ganesh C. Gopalakrishnan, David R. Smith, and Mandayam K. Srivas. An algebraic approach to the specification and realisation of VLSI designs. In *Computer Hardware Description Languages and their Applications*, pages 16–38. IFIP, August 1985.

[7] Mike Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design*, pages 153–177, 1986.

[8] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science. Prentice-Hall, 1985.

[9] Dieter Hogrefe. *Estelle, LOTOS und SDL: Standard-Spezifikationssprachen für Verteilte Systeme.* Springer-Verlag, 1989.

[10] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* ACM Distinguished Dissertations. MIT Press, 1984.

[11] Carlos Delgrado Kloos and Walter Dosch. Transformational development of digital circuit descriptions: a case study. In *Compeuro Conference on VLSI and Computers*, pages 319–322. IEEE, May 1987.

[12] Michael C. McFarland, Alice C. Parker, and Raul Camposano. Tutorial on high level synthesis. In *Design Automation Conference*, pages 330–336. ACM/IEEE, June 1988.

[13] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science. Springer-Verlag, 1980.

[14] Ben Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[15] John T. O'Donnell. Hardware description with recursion equations. In *Computer Hardware Description Languages and their Applications*, pages 363–382. IFIP, April 1987.

[16] James L. Peterson. *Petri net theory and the modeling of systems.* Prentice-Hall, 1981.

[17] Ranga Vemuri. How to prove the correctness of a set of register level design transformations. In *Design Automation Conference*, pages 207–212. ACM/IEEE, June 1990.

[18] William M. Waite and Gerhard Goos. *Compiler Construction.* Text and Monographs in Computer Science. Springer-Verlag, 1984.

Parallel Computers for Advanced Information Processing

ir E. Odijk
Philips Natuurkundig laboratorium
P.O. box 80.000, 5600 JA Eindhoven, The Netherlands
e-mail: odijk@prl.philips.nl

Abstract

Within the Computer Architecture Department of Philips Research Laboratories Eindhoven, a five years research effort has resulted in the design and implementation of a parallel object-oriented computer system, POOMA, and a parallel object-oriented language to express programs for execution on this system.

Conceptually, POOMA is a distributed-memory, scalable, computer that consists of self-contained computers, each with their own cpu, main memory and a communication unit. These computers communicate via a sparsely connected, packet-switching, topology of bidirectional communication links that are serviced by the communication unit.

The architecture (which is understood to denote the functionality as implemented by both the hardware and systems software) has been designed to meet the requirements of the parallel object-oriented language, and the match of the conceptual and operational models can be well demonstrated.

The presentation will include a survey of the systems concepts, the rationale of a number of the design choices and an informal presentation of the lessons that have been learnt during the design.

Contents

The contents of the article is printed in a separate handout. It is a copy of the publication in IEEE micro of december.

Parallel Discrete Event Simulation

Benno Overeinder Bob Hertzberger Peter Sloot
Department of Computer Systems
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
e-mail: `overeind@fwi.uva.nl`

Abstract

In simulating applications for execution on specific computing systems, the simulation performance figures must be known in a short period of time. One basic approach to the problem of reducing the required simulation time is the exploitation of parallelism. However, in parallelizing the simulation new problems arise. Due to the distributed generation of events causality errors can occur, as a result the sequence in which to process the events is essentially indeterminated.

In this paper we present a model to analyse the inherent parallelism of a simulation, together with a survey of existing strategies to perform the simulation in parallel. Some extensions to this model are discussed, resulting in reliable evaluation of the effectiveness of these strategies.

1 Introduction

In the Parallel Scientific Computing Working-group at the University of Amsterdam, we are interested in the execution performance of classes of applications on classes of computing systems. We distinguish the following levels that are involved in performance prediction: application, general abstract machine, simulation language, and discrete event simulator. Each level is supported by the level underneath. In this way the efficiency of a level is partially determined by the supporting level, thus imposing severe constraints to the simulator. Especially if the performance figures are iteratively used to optimize the application, the effectiveness of the simulator is of vital importance.

Large discrete event simulations are known to consume enormous amounts of time on sequential machines. One basic approach to reduce the required simulation time is the exploitation of parallelism. A major drawback however, is the inherent complexity of this type of simulation since the notion of global time does not easily map on a parallel computer. Sophisticated clock synchronization algorithms are required to ensure that cause-and-effect relationships are correct reproduced by the simulator.

The idea of parallel simulation—in literature also indicated by distributed simulation—was first proposed by K.M. Chandy and independently by R.E. Bryant. Papers by Chandy and Misra [Cha79], and Bryant [Bry77] contain basic ideas of parallel simulation, the problem of deadlock and schemes for deadlock resolution, detection and recovery [Cha81]. Alternative schemes proposed by D.R. Jefferson are based on the concepts of Virtual Time [Jef85].

This paper is structured in the following way. Section 2 gives an introduction to discrete event simulation. In section 3 a parallel view to the sequential simulation is proposed, and various methods for parallel simulation are described together with a discussion on their effectiveness. Finally, in section 4 an evaluation of these methods and some suggestions for further research are presented.

2 Concepts of Discrete Event Simulation

Modelling and simulation can be characterized as the complex of activities associated with constructing models of real world systems and simulating them on a computer.

Essential to every model is the time base on which events occur. Accordingly, models can be classified depending on their temporal behaviour [Zei76]. A model is a *continuous time* model when time flows smoothly and continuously. A model is a *discrete time* model if time flows in jumps of some specified time unit.

A second classification can be based on the range sets of a model's descriptive variables. The model is a *continuous state* model if the range of the descriptive variables can be represented by the real numbers. The model is a *discrete state* model if its variables only assume discrete values.

Continuous time models can be further divided into *differential equation* and *discrete event* classes. A differential equation model is a continuous time–continuous state model where changes in state occur smoothly and continuously in time. In a discrete event model, even though time flows continuously, state changes can occur only at countable points in time—i.e., time jumps from one event to the next, and these events can occur arbitrarily separated from each other.

2.1 Discrete Event Simulation

The concept of a system and a model of a system were already used in the definition of the classes of simulation. These concepts need to be specified in order to develop a framework for the design of a discrete event model of a system. The major concepts are:

System A collection of entities that interact together over time to accomplish one or more goals.

Model An abstract representation of the system under consideration, usually containing logical and/or mathematical relationships that describe the behaviour of the system.

System state A collection of variables that contain all the information necessary to describe the system at any time.

Entity Any object or component in the system that requires explicit representation in the model.

Attributes The properties of a given entity.

Event An instantaneous occurrence that may change the state of the system.

Activity A duration of time of specified length during which entities engage some operation.

Process A sequence of events ordered in time. These events must be logically connected, involving the same entity.

To illustrate these concepts, we consider a bank. In the dynamics of a bank, customers might be one of the entities, the balance in their accounts might be an attribute, and making deposits might be an activity. Possible state variables are the number of busy tellers, the number of customers waiting in line or being served, and the arrival time of the next customer. The arrival of a customer as well as the completion of service of a customer are possible events.

Every discrete event simulation contains a state variable called the *simulation clock* to model the flow of time. Simulated time is advanced from the time of the current event to the time of the next scheduled event; thus skipping periods of inactivity. Future events are stored in a calendar that contains the time and the type of all scheduled events, usually in chronological order. The nature of the routine depends on the world view used in the model. Let us therefore consider some different world views relevant to discrete event simulation.

2.2 World Views

All simulations contain an executive routine for the management of the calendar and clock, i.e., the sequencing of events and driving of the simulation. This executive routine fetches the next scheduled event, advances the simulation clock and transfers control to the appropriate routine. The operation routines depend on the world view, and may be events, activities, or processes.

A world view is the point of view from which the modeller sees the world or the system to be modelled. Most of the discrete event simulations use one of the three following perspectives [Hoo86]: *event scheduling*, *activity scanning*, or *process interaction*.

In *event scheduling* each type of event has a corresponding event routine. The executive routine processes a time ordered calendar of event notices to select an event for execution. Event notices consist of a time stamp and a reference to an event routine. Event execution can schedule new events by creating an event notice and place it at the appropriate position in the calendar. The clock is always updated to the time of the next event, the one at the top of the calendar.

In the *activity scanning* approach a simulation contains a list of activities, each of which is defined by two events: the start event and the completion event. Each activity contains test conditions and actions. The executive routine scans the activities for satisfied time and test conditions and executes the actions of the first selectable activity. When execution of an activity completes, the scan begins again.

The *process interaction* world view focuses on the flow of entities through a model. This strategy views systems as sets of concurrent, interacting processes. The behaviour of each class of entities during its lifetime is described by a process class. Process classes can have multiple entries and exits at which a process interacts with its environment. The executive routine uses a calendar to keep track of forthcoming tasks. However, apart from recording activation time and process identity, the executive routine must also remember the state in which the process was last suspended.

Evidently, large discrete event simulations, using one of these three world view strategies, put extreme computational demands on sequential computers. Intuitively, the process interaction world view seems to be attractive as a starting point in our effort to the parallelization of the simulation. The modeller perceives the simulation already as a set of concurrent objects interacting with each other by well-defined communication. Besides, parallel simulation is interesting because it represents a problem domain that often contains substantial amounts of inherent parallelism (e.g., see [Liv85]).

In the following section a parallel view to a sequential execution will be presented in order to analyse the inherent parallelism of the simulation. Next the problems involved in parallel execution and the methodologies to circumvent these problems are described.

3 From Sequential to Parallel Discrete Event Simulation

3.1 The Average Parallelism Measure

If we have made the decision to do the simulation in parallel, there are some fundamental questions to be answered. What is the parallelism inherent to the simulation? How much benefit do we expect from doing things in parallel? And, once the job is done, how well did we perform this?

One very interesting characterization of the simulation that can be used to answer these questions is the *average parallelism*. Average parallelism can be defined in two equivalent ways:

1. The ratio of the total service time required to process events, to the length of the critical path through the execution of the simulation.
2. The speedup figures, if a hypothetical machine contains an unbounded number of available processors and zero synchronization overhead.

As a consequence of the second definition, the average parallelism figure should be regarded as an upper bound to the speedup that can be achieved.

To reveal the average parallelism inherent to a simulation, we have implemented a tool to analyse a sequential simulation run and extract the average parallelism [Ove91]. A system model is defined to express the parallelism explicitly and consists of a software component and a hardware component. The software component is a graph representing the execution of a sequential simulation. The hardware component of our system model reflects our focus on the parallelism inherent to a simulation, and makes assumptions of ideal hardware.

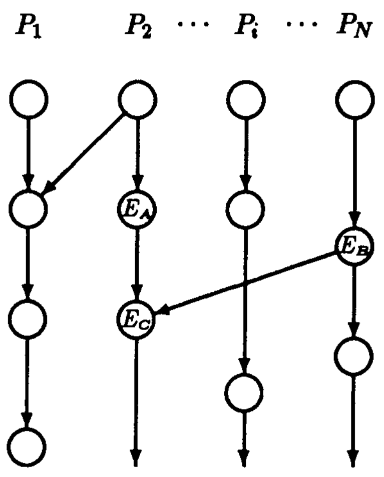

Figure 5: A program activity graph.

The execution of a simulation is represented by an acyclic directed graph (see Fig. 5). Each vertex of the graph corresponds to an event occurring in the simulation. Precedence constraints exist among the events, modelling the chronological order of events. These precedence constraints are modelled by the arcs of the graph: an arc from vertex E_A to vertex E_C means that event E_C cannot occur (or be executed) before event E_A is processed. Two types of arcs are distinguished: *intra-process arcs* and *inter-process arcs*. Intra-process arcs are precedence constraints between events that occur within the same process (e.g., arc between vertex E_A and E_C in Fig. 5). The intra-process arc denotes an independent unit of sequential work inside a process. We can consider inter-process arcs as precedence constraints between events that occur in different processes (e.g., arc between vertex E_B and E_C). These inter-process arcs represent synchronization requirements achieved by some communication primitive.

The hardware component of the system is modelled as an infinite number of identical processors, each of unit speed. The synchronization between processors has zero overhead and the entire computer is devoted to one single task.

A sequential run of the simulation generates an acyclic directed graph of events with their precedence constraints. When every process in the simulation is assigned to a different processor (i.e., one process to one exclusive processor), all *intra*-process dependent events occur at the same exclusive processor and all *inter*-process dependent events occur at different processors. As a consequence, the *intra*-process arc denotes an independent unit of sequential work on a processor, whereas the *inter*-process arc represents synchronization requirements between processors. Furthermore, the execution times of the independent units of work, measured during the sequential run, are assigned to the *intra*-process arcs and the zero synchronization costs to the *inter*-process arcs. In this way the graph is reduced to a representation of the execution of the simulation on a hypothetical machine. The total amount of time required to process the events is equal to the sum of all the costs in the graph and the critical path through the execution of the simulation is now represented by the longest path in the graph.

Eager et al. [Eag89] use the average parallelism measure to express lower bounds on speedup and efficiency, and on the incremental benefit and cost of allocating additional processors. It is our opinion that average parallelism can be applied as a measure in the evaluation of effectiveness of various methods in parallel simulation. In other words, how much of the parallelism that is inherent to the simulation is actually exploited?

3.2 The Fundamental Problem in Parallel Discrete Event Simulation

We are especially interested in parallelization of asynchronous system simulation, where events are not synchronized by a global clock, but rather occur at irregular time intervals. In these simulations few events occur at any single point in simulated time and therefore parallelization techniques based on synchronous execution using a global simulation clock performs poorly. Concurrent execution of events at different points in simulated time is required, but this introduces interesting synchronization problems.

These problems become clear if one examines the operation of a sequential discrete event simulator. The sequential simulator typically uses three data structures: the state variables, an event list (the calendar), and a global simulation clock. For the execution routine (see section 2.2) it is crucial that the smallest time stamped event (E_{min}) from the event list is selected as the one to be processed next. If it would depart from this rule and select an other event with a larger time stamp (E_x), it would be possible for E_x to change the state variables used by E_{min}. This implies that one is simulating a system where the future could affect the past. We call errors of this kind *causality errors.*

Let us next consider the parallelization of a simulation based on the above paradigm. Most parallel discrete event simulation (PDES) strategies adhere to a process interaction world view that strictly forbids processes to have direct access to shared state variables. To this methodology some extensions have been made to support the parallel execution of the simulation [Cha79]. The system being modelled is viewed as being composed of some number of *physical processes* that interact at various points in simulated time. The simulation is constructed as a set of *logical* *processes* LP_0, $LP_1, \ldots$, one per physical process. All interactions between physical processes are modelled by time stamped event messages sent between the corresponding logical processes. Each logical process contains a portion of the state corresponding to the physical process it models, as well as a local clock that denotes the progress of the process.

One can assure that no causality error occurs if one adheres to the local causality constraint:

> **Local Causality Constraint:** A discrete event simulation, consisting of logical processes that interact exclusively by exchanging time stamped messages, obeys the local causality constraint *if and only if* each logical process executes events in non decreasing time stamp order.

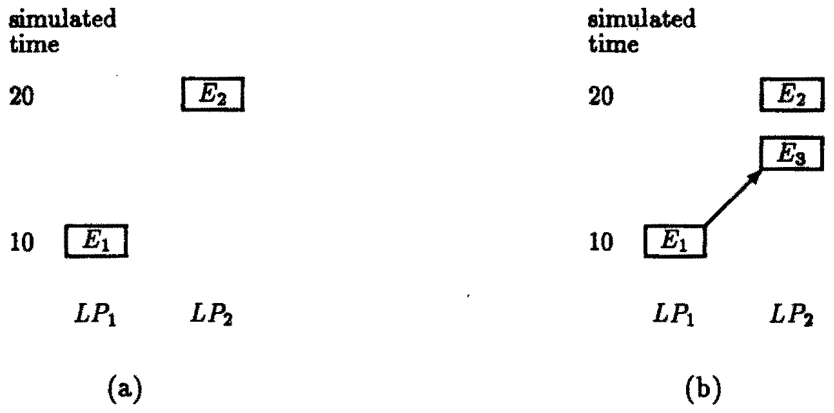

Figure 6: Causality error.

Consider two events. E_1 at logical process LP_1 with time stamp 10, and E_2 at LP_2 with time stamp 20 (see Fig. 6). If E_1 schedules a new event E_3 for LP_2 containing a time stamp less than 20, then E_3 could affect E_2, necessitating sequential execution of all three events. If one had no information what events could be scheduled by other events, one would be enforced to process the only save event, the one containing the smallest time stamp, resulting in a sequential execution.

During the simulation we must therefore decide whether E_1 can be executed concurrently with E_2. But how do we know whether or not E_1 affects E_2 without actually performing the simulation for E_1? It is this question the parallel discrete event simulation strategies must address.

In this paper we classify parallel discrete event simulation strategies by two categories: *conservative* and *optimistic*. Conservative approaches strictly avoid the possibility of any causality error ever occurring. These approaches rely on some strategy to determine when it is safe to process an event. The optimistic approaches use a detection and recovery approach: whenever causality errors are detected a rollback mechanism is invoked to recover. We will describe some of the concepts behind conservative and optimistic simulation mechanisms.

3.3 Conservative Methods

The conservative approaches are the first distributed simulation mechanisms. The basic problem conservative mechanisms must address is to determine which event is save to process. If a process contains an event E_1 with time stamp T_1 and the process can determine that it is impossible to receive another event with time stamp smaller than T_1, then the process can safely process event E_1 without a future violation of the local causality constraint. Processes containing no safe events must block; this can lead to deadlock situations if no appropriate precautions are taken.

Independently, Chandy and Misra [Cha79], and Bryant [Bry77] developed the parallel discrete event simulation algorithms, where one statically specifies the links that indicate which process may communicate with which other processes. In order to determine when it is safe to process a message, it is required that messages from any process to any other process are transmitted in chronological order according their time stamps. Each link has a clock associated with it that is equal to either the time stamp of the message at the front of that link's queue or, if the queue is empty, the time of the last received message. The process repeatedly selects the link with the smallest clock and, if there is a message in that link's queue, updates its local clock to the link's clock and process the message. The order of event processing will be correct because all future messages received will have later time stamps than the local clock, since they will arrive in chronological order along each link. If the selected queue is empty, the process blocks. This is because the process may receive a message over this link with a time that is less than all the other input time stamps. Thus to insure correct chronology, the process is forced to wait for a message to update the clock on the link before the process can update its local clock. This protocol guarantees that each process will only process events in nondecreasing time stamp order, and thereby ensuring chronological integrity.

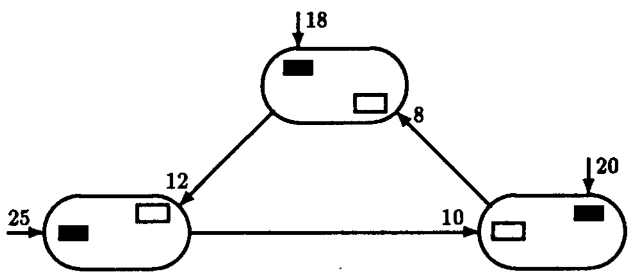

Figure 7: An example of deadlock. (The numbers indicate time stamps.)

Deadlock occurs when there is a cycle of blocked processes and each process is blocked due to another process in the cycle. For example consider the network of Fig. 7. Each process is waiting on the incoming link containing the smallest clock value because the corresponding queue is empty. All three processes are blocked, even though there are event messages in other queues that are waiting to be processed.

Null messages are used to avoid deadlock. This scheme requires that there is a strictly positive lower bound on the *lookahead* for at least one process in each cycle. Lookahead is defined to be the amount of time that a process can look into the future. In other words, if the local clock of the process is any time T and the process can predict all messages it will send with time stamps less than $T + L$, where L is the lookahead. Thus, for a queueing network model, a strictly positive lower bound for the service time for some stations would be required. Intuitively, processes keep the clocks of their output links ahead of their local clocks by sending null messages. A null message with time stamp T_{null} from process LP_A to LP_B, tells LP_B that there will be no more messages from process LP_A with time stamp less than T_{null}. Whenever a process finishes processing an event, it sends a null message on each of its output ports indicating the lower bound on the time stamp of the next outgoing message. The receiver of the null message can then compute new bounds on its outgoing links, send this information to its neighbours, and so on.

Chandy and Misra [Cha81] also presented a two-phase scheme where the simulation proceeds until deadlocked, then the deadlock is detected and resolved. The mechanism is similar to that described above, except no null messages are created. Instead the computation is allowed to deadlock. The scheme involves a controller process to monitor for deadlock and control deadlock recovery. Deadlock detection mechanisms are described in [Gro89, Mis86]. The deadlock can be broken by the observation that the message with the smallest time stamp is always save to process; or, with use of a distributed computation, obtain a lower bound to enlarge the set of safe messages.

The mechanisms described above only attempt to detect and recover from global deadlocks. Prakash and Ramamoorthy [Pra88] suggested a hierarchical decentralized algorithm that takes advantage of the locality of these deadlocks. Another approach to detect and recover from local deadlocks can be found in [Mis86].

The performance of conservative mechanisms is critically determined by the degree to which processes can look ahead and predict future events; or more importantly, what will not happen in the simulated future. A process with lookahead L can guarantee that no events, other than the ones that it can predict, will be generated up to time Clock $+ L$. This may enable processes to safely process forthcoming messages that they have already received. Fujimoto describes lookahead quantitatively using a parameter called the lookahead ratio and presents empirical data to demonstrate the importance of exploiting lookahead to achieve good performance [Fuj89]. Other studies of the performance as a function of lookahead can be found in [Lin89, Lou90, Su89].

3.4 Optimistic Methods

In optimistic approaches a process's clock may run ahead of the clocks of its incoming links and if errors are made in the chronology a procedure to recover is invoked. In contrast to conservative approaches, optimistic strategies need not determine when it is safe to proceed. Advantages of this approach are that it has a potentially larger speedup than conservative approaches and that the topology of possible interactions between processes need not be known.

An optimistic approach to distributed simulation called Time Warp, based on the Virtual Time paradigm, was proposed by Jefferson and Sowizral [Jef82, Jef85]. Here virtual time is the same as the simulated time. The local clock, called the Local Virtual Time (LVT) of a process, is set to the minimum receive time of all unprocessed messages. Processes can execute events and proceed in local simulated time as long as they have any input at all. As a consequence, the local clock or LVT of a process may get ahead of its predecessors' LVTs, and it may receive an event message from a predecessor with time stamp smaller than its LVT, i.e., in the past of the process. If this happens the process *rolls back* in simulated time. The event causing the roll back is called a *straggler*. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler.

The premature execution of an event results in two things that have to be rolled back: the state of the logical process and the event messages to other processes. Rolling back the state is accomplished by periodically saving the process state and restoring an old state vector on roll back. Unsending a previously sent message is accomplished by sending a *anti-message* that annihilates the original when it reaches its destination. Messages that are sent while the process is propagating forward in simulated time are called *positive messages*. If a process receives an anti-message that corresponds to a positive message that is still in the input queue, then the two will annihilate each other and the process will proceed. If an anti-message arrives that correspond to a positive message that is already processed, then the process has made an error and must also roll back. It sets its current state to the last state vector saved with simulated time earlier than the time stamp of the message. A direct consequence of the roll back mechanism is that more anti-messages may be sent to other processes recursively.

The Global Virtual Time (GVT) is the minimum of the LVTs for all the processes and the time stamps of all messages sent but unprocessed. No event with time stamp smaller than GVT will ever be rolled back, so storage used by such event (i.e., saved states) can be discarded.

The procedure just described is referred to as Time Warp with aggressive cancellation. An alternative is lazy cancellation, where anti-messages are not sent immediately after roll back. Here, the process resumes executing forward in simulated time from its new LVT, and when it procedures a message it compares it with the messages in its output queue. If the same message is recreated, then there is no need to cancel the message. An anti-message created at simulated time T is only sent after the process's clock sweeps past time T without regenerating the same message. Thus, under lazy cancellation a roll back at the successor process may be avoided. On the other hand, if messages are not reproduced, then roll backs at the successor processes will be required under both mechanisms, and they will occur sooner with aggressive cancellation.

Depending on the application, lazy cancellation may either improve or degrade performance. States may be saved less frequently at the expense of greater overhead for roll back. As a consequence, lazy cancellation requires more memory than aggressive cancellation. Studies of the performance of optimistic approaches can be found in [Lin90, Mad90].

4 Conclusion and Discussion

Performance evaluation is critical for the design, implementation, and improvement of complex applications executing on parallel computers. Analytical approaches to performance evaluation are usually inadequate because they are based on unrealistic assumptions and require many approximations. Therefore, simulation is a good alternative for obtaining accurate measures of performance. Currently, however, detailed simulations are extremely slow. Parallel simulation seems to be a promising approach for speeding up the simulations, although much more work needs to be done to increase the effectiveness of the existing methods.

Conservative methods offer good potential for certain classes of problems. A major drawback, however, is that they cannot fully exploit the parallelism available in the simulation application. If it is possible that event E_A might affect E_B either directly or indirectly, conservative approaches must execute E_A and E_B sequentially. If the simulation is such that E_A seldom affect E_B these events could have been processed concurrently most of the time. As a consequence, conservative algorithms heavily rely on lookahead to achieve good performance.

Optimistic methods offer the greatest potential as a general purpose simulation mechanism. A critical question faced by optimistic approaches is whether the system will spent most of its time on executing incorrect computations and rolling them back, at the expense of correct computations. An intuitive explanation why the behaviour tends to be stable is that incorrect computations can only be initiated by a premature execution of a correct event. This premature execution, and subsequent incorrect computations, are by definition in the simulated time future of the correct, straggler computation. Also, the further the incorrect computation spreads the further it moves into the simulated time future, thus lowering its priority for execution. Preference is always given to computations containing smaller time stamps. The incorrect computation will be slowed down, allowing the error detection and correction mechanism to correct before too much damage has been done.

A more serious problem with the optimistic mechanisms is the need to periodically save the state of each logical process. This limits the effectiveness of the optimistic mechanisms to applications where the amount of computation, required to process an event, is significantly larger than the cost of saving the state vector.

The type of application, or classes of applications, is important when determining an appropriate approach to distributed simulation. For dynamic topology systems and systems with irregular interactions, Time Warp methods are preferred over conservative methods, especially if state-saving overheads do not dominate. On the other hand, if the application has good lookahead properties, conservative algorithms can exploit the special structure within a fixed topology system. If the application has both poor lookahead and large state-saving overheads all existing parallel discrete event simulation approaches will have trouble obtaining good performance, even if the application has a considerable amount of parallelism.

A challenging, yet not fully exploited, problem is the use of hierarchical methods in parallel discrete event simulation (PDES). It is our contention that, if processes are forced to remember the values of all private variables, an object-oriented methodology can be employed. Here a class must encapsulate all relevant aspects of an entity: its attributes, actions, and life cycle. Communication between objects is allowed only through well-defined interfaces, described by the types of messages an object is willing to respond to. With the use of such object-oriented methodologies, the hierarchical decomposition of the problem under investigation can also be made available in the simulation. In conservative approaches there is some modest effort to use this hierarchical knowledge in the detection of local deadlock and recovery [Pra88]. In optimistic approaches, hierarchical knowledge could be used by the error detection and correction mechanism to quickly stop the spread of the erroneous computations. Furthermore, the proposed model in section 3.1 has to be extended for the evaluation of the various PDES strategies. Many performance evaluations of PDES strategies, found in the literature, compare the parallelism available in the application with the measured speedup of the application on a specific parallel computer. In consequence, there is interference with load balance and scheduling strategies that obscure the effectiveness of the PDES strategy. The extended model should eliminate this interference, and measure the exploited parallelism by a PDES strategy. In this way, the exploited parallelism can be compared to the average parallelism to obtain the effectiveness of the strategy.

Acknowledgements

I would like to thank Sjaak Koot from our working-group for some valuable discussions.

References

[Bry77] Bryant, R.E., "Simulation of Packet Communications Architecture Computer Systems," MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[Cha79] Chandy, K.M., and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, September 1979.

[Cha81] Chandy, K.M., and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, no. 11, pp. 198–205, November 1981.

[Eag89] Eager, D.L., J. Zahorjan, and E.D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," *IEEE Transactions on Computers*, vol. 38, no. 3, pp. 408–423, March 1989.

[Fuj89] Fujimoto, R.M., "Performance Measurements of Distributed Simulation Strategies," *Transactions of the Society for Computer Simulation*, vol. 6, no. 2, pp. 89–132, April 1989.

[Gro89] Groselj, B., and C. Tropper, "A Deadlock Resolution Scheme for Distributed Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 108–112, March 1989.

[Hoo86] Hooper, J.W., "Strategy Related Characteristics of Discrete Event Languages and Models," *Simulation*, vol. 46, no. 4, pp. 153–159, April 1986.

[Jef82] Jefferson, D.R., and H. Sowizral, "Fast Concurrent Simulation using the Time Warp Mechanism, Part I: Local Control," Technical Report N-1906-AF, RAND Corporation, December 1982.

[Jef85] Jefferson, D.R., "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, July 1985.

[Lin89] Lin, Y-B., and E. Lazowska, "Exploiting Lookahead in Parallel Simulation," Technical Report 89-10-06, Department of Computer Science, University of Washington, Seattle (WA), 1989.

[Lin90] Lin, Y-B., and E. Lazowska, "Reducing the State Saving Overhead for Time Warp Parallel Simulation," Technical Report 90-02-03, Department of Computer Science, University of Washington, Seattle (WA), 1990.

[Liv85] Livny, M., "A Study of Parallelism in Distributed Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 94–98, San Diego (CA), January 1985.

[Lou90] Loucks, W.M., and B.R. Preiss, "The Role of Knowledge in Distributed Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 9–16, San Diego (CA), January 1990.

[Mad90] Madisetti, V., J. Walrand, and D. Messerschmitt, "Synchronization in Message-Passing Computers—Models, Algorithms, and Analysis," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 35–48, San Diego (CA), January 1990.

[Mis86] Misra, J., "Distributed Discrete Event Simulation," *ACM Computing Surveys*, vol. 18, no. 1, pp. 39–65, March 1986.

[Ove91] Overeinder, B.J., and P.M.A. Sloot, "Parallelism in Architecture Simulation," Technical Report, Department of Computer Systems, University of Amsterdam, Amsterdam, The Netherlands, under preparation.

[Pra88] Prakash, A., and C.V. Ramamoorthy, "Hierarchical Distributed Simulations," *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 341–347, San Jose (CA), June 1988.

[Su89] Su, W.K., and C.L. Seitz, "Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 38–43, March 1989.

[Zei76] Zeigler, B.P., *Theory of Modelling and Simulation*, John Wiley & Sons, New York, 1976.

A Partitioning and Redundancy Model for Wafer-Scale Integrated Circuits.

Dr.ir. Martin F. Beusekamp
University of Twente
Department of Computer Science
P.O. Box 217
7500 AE Enschede
Telefoon: 053 - 893796
Telefax: 053 - 356531

Abstract

In wafer scale integration, a certain amount of redundancy is required to achieve an acceptable yield. In general, this redundancy is added to the circuit after the step of implementing the architecture, by duplicating specific parts of the circuit and providing for the necessary wiring and switches to reconfigure the circuit after testing. However, this is not the optimal procedure. This paper presents a general, architecture-independent model to calculate the required amount of redundancy and the necessary degree of partitioning of the circuit to achieve a maximum efficiency. The model takes several technological parameters into account. It is shown that in practical cases, the efficiency curve around the maximum is relatively flat.
The model presents the optimal degree of partitioning, rate of redundancy, total area, yield and efficiency for any desired (large) circuit. It is shown that, for every technological environment, there exists a critical circuit size, below which the implementation of partitioning and redundancy is undesirable. Above this critical circuit size, partitioning into rather small cells leads to optimal efficiency, in spite of the fact that small cells require a considerable overhead of reconfiguration wiring and switches. This also means that the implementation of very large redundant cells or blocks will cost more area than the enhancement of efficiency justifies. It was shown that the model can be adapted to any specific architecture by adjusting parameters and limiting possible solutions to restricted, architecture-dependent values. By a simple mathematical substitution, the model can be made completely analytic, making it very simple to investigate the sensitivity of the optimal solution to small variations in parameters.

Contents.

List of symbols.

The following symbols will be used in this memorandum. In the text, the first occurrence of a symbol is printed in boldface.

a	$= A_c/A_{c,50}$
A_c	= area of desired circuitry, if processed in a conventional way
$A_{c,50}$	= area of circuitry which will show a yield of 50% in the given technological process
a_{crit}	= circuit area from which the implementation of partitioning and redundancy becomes favourable
A_r	= redundant area
A_t	= total area
A_w	= area needed for additional wiring and reconfiguration switches
$A_{w,50}$	= area of wiring which will show a yield of 50% in the given technological process
b	$= A_{w,50}/A_{c,50}$
E	= efficiency $Y.(A_c/A_t)$
E^*	= preliminary efficiency $Y.(k/n)$
E_{max}	= maximum efficiency at a given value of k
i	= general integer variable
k	= number of required basic cells
n	= number of available basic cells
P	= general variable of probability
w	= parameter characterizing the overhead of additional wiring and reconfiguration switches
$y, y(k)$	= single-cell yield
Y	= overall yield
y_c	= yield of circuit part of a single cell
y_w	= yield of wiring part of a single cell

1. Scope of this memorandum.

The past few decades have shown a rapidly increasing level of the integration of electronic circuits in monolithic silicon devices. A better understanding of and control over technological processes and the development of cleaner fabrication facilities have led to chip sizes as large as several cm^2 and to smallest feature sizes in the order of 1 μm, with 0.7 μm predicted in the near future [1]. However, physical limits will make it hardly possible to continue this increase of integration. Firstly, because it will be very expensive to construct clean-room fabrication facilities with a significantly lower dust-particle density than the present state-of-the-art (class 10, indicating a maximum of 10 dust particles >0.5 μm per cubic foot (28 dm^3) of air). Secondly, because a decrease in the smallest feature size to, say, a few tenths of a μm, requires tools and equipment beyond the present technological possibilities [2].

Nevertheless, there is a great demand for large electronic circuits, integrated in one package, mainly because of the higher achievable speed and reliability. Chip-on-board and other hybrid techniques provide some possibilities, but realization on a single silicon wafer will yield smaller and cheaper circuits with a higher performance and a significantly higher reliability. To be able to produce very large monolithic silicon devices, the traditional policy of immediate disposal of defective devices is abandoned. Techniques to repair these devices are in development. Moreover, the concept of redundancy is widely accepted as a means to provide "spare parts", substituting defective blocks of circuitry on chip. Of course, this requires the possibility of rewiring circuits on chip, for which methods are in development as well. They range from purely physical, irreversible techniques like laser cutting and laser fusion to software controlled, reversible semiconductor switches. The former method has the advantage of requiring hardly any additional chip area, the latter allows implementations which are reconfigurable for an infinite number of times, if necessary per algorithm. An overview is given in [3].

Various techniques aiming for the integration on monolithic silicon wafers of circuits which are significantly larger than state-of-the-art VLSI-technology allows, are known under the general term "Wafer Scale Integration". This memorandum will discuss a mathematical model, based on physical reality, concerning the partitioning of large circuits into smaller circuits and the addition of redundancy in wafer scale integrated circuits.

2. Introduction.

The basic limiting factor in increasing the chip size on a silicon wafer is, of course, the yield. The main problem is the high probability that a relatively large circuit is disturbed by a dust particle. Increasing the chip size will dramatically decrease the probability of producing a significant number of faultless chips, as the probability of faults increases exponentially with the area.

In Wafer-Scale Integration (WSI), redundancy is used to implement circuit sizes which are significantly larger than state-of-the-art VLSI-techniques permit, and still

achieve an acceptable yield. Many models [e.g. 4,5,6] have been published to calculate and optimize the effect of redundancy in more or less specific architectures. This reflects the general policy of implementing redundancy by duplicating certain parts of the micro-architecture of the circuit under design, without paying appropriate attention to the actual sizes of the duplicated sub-circuits and the amount of redundancy which is implemented by this duplication [e.g. 7,8].
This memorandum, however, will present an architecture-independent model to calculate the optimal partitioning and the required amount of redundancy to achieve a maximum efficiency. It takes several technological parameters into account. At the end, it is indicated how this model can be adapted to any specific architecture. In the appendix, a mathematical substitution is presented, which makes the model completely analytical. This means that by derivation with respect to its parameters, the influence of these parameters can very easily be investigated.

3. The partitioning and redundancy model.

3.a. General assumption.

The main purpose of the model to be presented in this memorandum, will be to gain insight in the optimal rate of partitioning *any* desired circuit into sub-circuits and the application of such an amount of redundancy that a maximum efficiency is reached. In order to derive an architecture-independent model it will, for the time being, be assumed that the desired (large) circuit can be partitioned into any (positive integer) number **k** of identical cells, each having a probability **y** of functioning correctly. In general, this will be an unrealistic assumption, but in section 4 it will be shown that the model can easily be adapted to any specific, realistic architecture by merely adjusting some parameters and limiting the possible solutions to restricted, architecture-dependent values.

3.b. The concept of partitioning and redundancy.

In case of a conventional chip without redundancy, consisting of k identical cells with a probability y of functioning correctly and with randomly distributed defects on the wafer, the overall yield **Y** will obviously be

$$Y = y^k \qquad (1)$$

which will be low for k more than a few units, unless y is very close to unity. This is illustrated by figures 1 and 2.

From expression (1) it is clear that, even with a moderately low number of required cells of, for instance, $k = 16$ and a relatively high single-cell yield of $y = 0.8$, the overall yield will only be

$$Y = (0.8)^{16} = 0.028. \qquad (2)$$

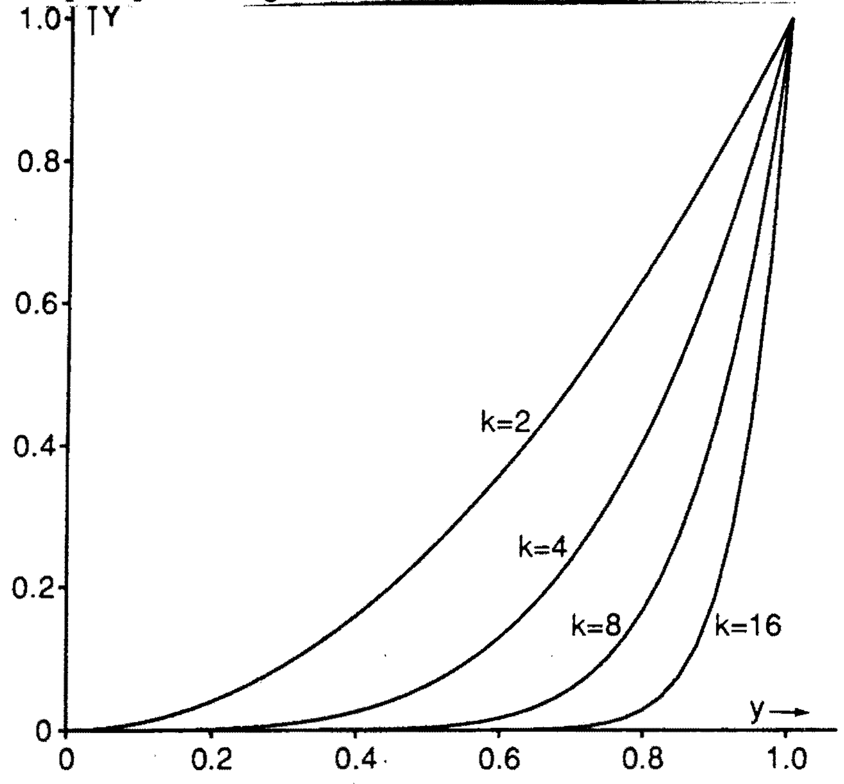

Figure 1. Overall yield Y of a conventional chip as a function of the single-cell yield y, with the required number k of cells as a parameter.

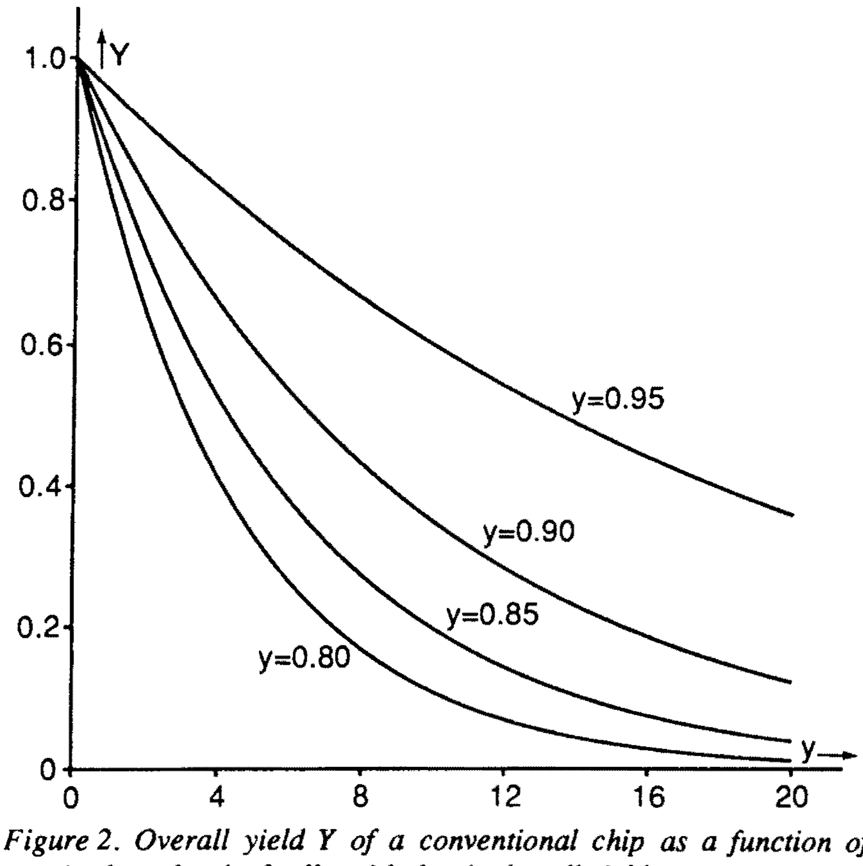

Figure 2. Overall yield Y of a conventional chip as a function of the required number k of cells, with the single-cell yield y as a parameter.

The effect of redundancy can easily be shown by expansion of this numerical example. Suppose that two of the above-mentioned structures were made, each consisting of 16 basic cells. The probability of finding at least one good structure will than be

$$Y = 1 - [1-(0.8)^{16}]^2 = 0.056 \tag{3}$$

This is still a low yield, mainly because the total of 32 basic cells have already been split into two independent sets of 16, irrespective of their usability. If the complete set of 32 cells is considered as one stock from which 16 working cells have to be selected, the overall yield can be calculated as follows.

Let us assume that a stock of **n** ($n \geq k$) of the above-mentioned basic cells is available. The probability **P** of finding *exactly* **i** faulty cells among these n cells is well known from basic theory on statistics

$$P = \binom{n}{i} y^{n-i} (1-y)^i \tag{4}$$

If k cells are required, a number of 0...n-k faulty cells is acceptable. Therefore, the probability of finding *at least* k good cells in a stock of n cells will be [9]

$$Y = \sum_{i=0}^{n-k} \binom{n}{i} y^{n-i} (1-y)^i \tag{5.1}$$

$$= \sum_{i=0}^{n-k} \frac{n!\, y^{n-i} (1-y)^i}{(n-i)!\, i!} \tag{5.2}$$

$$= n! \sum_{i=0}^{n-k} \frac{y^{n-i} (1-y)^i}{(n-i)!\, i!} \tag{5.3}$$

In the example of k = 16, n = 32 and y = 0.8, this leads to an overall yield of

$$Y = 32! \sum_{i=0}^{16} \frac{(0.8)^{32-i} (0.2)^i}{(32-i)!\, i!} = 0.99997 \tag{6}$$

which means that it is almost certain that at least 16 properly functioning cells can be found among the complete set of 32. This numeric example shows the strength of the concept of partitioning a circuit into smaller sub-circuits and applying redundant circuitry.

3.c. Illustration with numerical examples.

Let us first investigate expression (5) a little further by examining a few examples. An available set of n = 32 and a required set of k = 16 units show a strong dependency of the overall yield Y with respect to the single-cell yield y, but at least, y is in a very reasonable region

y	Y
0.2	0.00014
0.3	0.01384
0.4	0.16480
0.5	0.56997
0.6	0.90803
0.7	0.99476
0.8	0.99997

Table 1. Illustration of the strong influence of the single-cell yield y on the overall yield Y for n = 32 and k = 16.

This dependency becomes even stronger if the n/k-ratio is maintained, but the magnitudes of n and k are increased to n = 256 and k = 128

y	Y
0.4	0.00075
0.5	0.52491
0.6	0.99951

Table 2. Illustration of the stronger dependency between the single-cell yield y and the overall yield Y for larger values of n = 256 and k = 128.

Expression (5) also shows that even with a small amount of redundancy, an overall yield of Y = 0.9 can be achieved with reasonable values for the single-cell yield y

k	y
4	0.197
8	0.340
12	0.472
16	0.596
20	0.713
24	0.823
28	0.922

Table 3. The required single-cell yield y to achieve an overall yield of Y = 0.9 for various numbers k of required cells from a stock of n = 32 cells.

Finally, it can be shown that with a single-cell yield of y = 0.8 and a required overall-yield of Y = 0.9, the necessary amount of redundancy n/k deceases somewhat with an increasing number k of required cells

k	n	n/k
2	4	2.000
4	6	1.500
8	12	1.500
16	23	1.438
32	44	1.375
64	86	1.344
128	168	1.312

Table 4. Required amount of redundancy for various rates of partitioning.

It is clear that with a high number n of available cells, the overall yield Y will increase. But of course, the chip area needed to accommodate these n cells will increase as well and is in fact proportional to n. Therefore, an efficiency coefficient E* can be defined as the yield divided by n/k, the latter being the "penalty" which has to be paid for implementing redundant circuitry

$$E^* = Y.(k/n) = k(n-1)! \sum_{i=0}^{n-k} \frac{y^{n-i}\ (1-y)^{i}}{(n-i)!\ i!} \qquad (7)$$

For $n \to \infty$, it is clear that $Y \to 1$, but $E^* = Y.(k/n) \to 0$. For $n < k$ there is no solution possible, so $E^* = Y = 0$. Therefore, there must be a maximum in E* for $k \leq n < \infty$. This maximum can be found with a simple search algorithm which finds the optimal n for a given k and y. Some numerical examples are shown in tables 5 and 6.

y	n	Y	n/k
0.3	67	0.8920	4.19
0.4	50	0.9045	3.13
0.5	39	0.9002	2.44
0.6	32	0.9080	2.00
0.7	27	0.9202	1.69
0.8	23	0.9285	1.44
0.9	20	0.9568	1.25

Table 5. Optimal number n of cells, leading to a maximum efficiency E, for a given number of required cells of k = 16 and for various single-cell yields y.*

3.d. Extension to realistic WSI-circuits.

Obviously, the model as derived so far is not realistic enough, as it favours partitioning the desired circuit into an infinite number of very small cells. The reason for this is that partitioning a circuit into smaller parts requires additional wiring, which will increase the total area and decrease the overall yield, as wiring can fail as well. Moreover, the implementation of redundancy requires additional circuitry to reconfigure the circuit after testing. Therefore, the model must be extended to take this extra wiring and circuitry into account.

k	n	Y	n/k
2	2	0.6400	1.00
4	6	0.9011	1.33
8	12	0.9274	1.33
16	23	0.9285	1.30
32	46	0.9696	1.30
64	89	0.9757	1.28
128	174	0.9844	1.26

Table 6. Optimal number n of cells, leading to a maximum efficiency E, for a given single-cell yield of y = 0.8 and for various numbers k of required cells.*

Let $A_{c,50}$ denote that specific area of circuitry which will have a yield of 0.5 or 50% in the given technological process. $A_{c,50}$ is, of course, directly related to the average defect density of the process. Let A_c denote the area that would be required to integrate the desired circuit in a conventional way, without partitioning and/or redundancy. Let a be the ratio between A_c and $A_{c,50}$

$$a = A_c/A_{c,50} \tag{8}$$

Clearly, the overall yield Y of such a circuit, processed in the conventional way, would then be

$$Y = (0.5)^a = 2^{-a} \tag{9}$$

Let us now partition the desired circuit into k identical cells. Each of these cells will then have a single-cell yield of

$$y_c = 2^{-a/k} \tag{10}$$

Partitioning a circuit into *two* identical sub-circuits will generally introduce a "lane" between them, required for additional wiring and reconfiguration switches. It will therefore be realistic to assume that partitioning a circuit into k smaller sub-circuits will require an additional area A_w for wiring and reconfiguration switches which is proportional to ${}^2\log(k)$

$$A_w = w.{}^2\log(k).A_c \tag{11}$$

This means that every step of partitioning doubling the total number of required sub-circuits will add an equal amount of re-wiring area, captured in the parameter w. In practice, values for w of 0.1...0.5 may be expected. The inclusion of this re-wiring area will lead to a new total area needed for the circuit

$$A_c + A_w = [1 + w.{}^2\log(k)].A_c \tag{12}$$

The area of the wafer which is only occupied by additional wiring and switches will generally have a higher yield than the same area of circuitry, because fewer critical technological processing steps have been performed on it. Let $\mathbf{A_{w,50}}$ be that area for wiring and switches with a yield of 0.5 or 50% and let

$$\mathbf{b} = A_{w,50}/A_{c,50} \tag{13}$$

In practice, b will be in the order of 1...2. As a consequence, each of the k basic cells will require an additional amount of wiring and switches with a yield of

$$\mathbf{y_w} = (0.5)^{w.\,{}^2\log(k).a/bk} \tag{14.1}$$

$$= 2^{-(wa/bk).\,{}^2\log(k)} \tag{14.2}$$

$$= k^{-wa/bk} \tag{14.3}$$

resulting in a total yield for each cell of

$$\mathbf{y(k)} = y_c y_w \tag{15.1}$$

$$= 2^{-a/k}.k^{-wa/bk} \tag{15.2}$$

$$= (2.k^{w/b})^{-a/k} \tag{15.3}$$

Introducing redundancy can easily be done by substituting the constant y in (5.3) by the k-dependent y(k) of (15.3). The redundant area $\mathbf{A_r}$ of the complete circuit will be n/k - 1 times the area needed for circuitry, additional wiring and reconfiguration switches. Therefore, the total area occupied will now be

$$\mathbf{A_t} = A_c+A_w+A_r \tag{16.1}$$

$$= (n/k).(A_c+A_w) \tag{16.2}$$

$$= [1 + w.{}^2\log(k)].A_c.(n/k) \tag{16.3}$$

The efficiency **E** now has to be defined as the yield divided by the area penalty which has to be paid for implementing redundant circuitry and additional wiring plus reconfiguration switches

$$E = Y.(A_c/A_t) \tag{17}$$

With y(k) as in (15.3), this leads to an efficiency

$$E = \frac{n!\sum_{i=0}^{n-k}\dfrac{y(k)^{n-i}\,[1-y(k)]^i}{(n-i)!\;i!}}{[1 + w.{}^2\log(k)].(n/k)} \tag{18.1}$$

$$= \frac{k.(n-1)!}{[1 + w.{}^2\log(k)]}\sum_{i=0}^{n-k}\frac{y(k)^{n-i}\,[1-y(k)]^i}{(n-i)!\;i!} \tag{18.2}$$

For every k, there exists an n ($k \leq n < \infty$) for which the efficiency E shows a maximum E_{max}. The problem is to find the maximum of the maxima, for varying k, given the technological parameters w and b and the circuit size a. This can be done by a simple search algorithm. The result will be an optimal rate of partitioning, the required redundancy n/k to achieve the maximum efficiency E_{max}, the yield Y which will be achieved at maximum efficiency and the required total area A_t including the area needed for redundancy and re-wiring.

3.e. Results of the model.

Let us investigate expression (18) by assuming the technological parameters to have constant and realistic values of w = 0.3 and b = 1.5

a	k	n	n/k	Y	A_t/A_c	E_{max}
0.125	1	1	1.00	0.9170	1.00	0.9170
0.25	1	1	1.00	0.8409	1.00	0.8409
0.5	1	1	1.00	0.7071	1.00	0.7071
1.0	1	1	1.00	0.5000	1.00	0.5000
2	26	32	1.23	0.9670	2.97	0.3260
4	62	74	1.19	0.9812	3.33	0.2950
8	135	158	1.17	0.9863	3.66	0.2698
16	312	358	1.15	0.9938	4.00	0.2485
32	673	763	1.13	0.9955	4.33	0.2300

Table 7. Optimal rates of partitioning and redundancy for various amounts of required circuitry (a), at given values for the technological parameters w = 0.3 and b = 1.5

It can be seen that (in this case) for $a \leq 1$ both partitioning of the circuit as well as application of redundancy are not yet required and in fact undesirable, as the maximum efficiency is achieved for k = n = 1. For $a \geq 2$, a rather large and increasing rate of partitioning (k) leads to optimal results, with a gradually decreasing rate of redundancy (n/k). As could be expected, the required total area A_t/A_c increases, and the overall-efficiency E decreases with increasing a. It is surprising, however, that for circuit sizes ranging from a = 2 to a = 32, the maximum efficiency only decreases a factor of 1.42 (0.3260/0.2300).

The fact that for small values of a the application of redundancy is undesirable can be explained by looking at a graph of E as a function of n (or n-k), for various k as depicted in figure 3.

Obviously, for small values of k, meaning large basic cell areas, the area penalty which has to be paid by including redundancy is more severe than the enhancement of the yield which will be achieved. This also explains the behaviour of Y as a function of a. For relatively small circuits ($a \leq 1$), no redundancy is implemented and the yield Y will obviously decrease with increasing circuit size a. For larger circuits, the application of partitioning and redundancy causes an important enhancement of the yield, at maximum efficiency, at the cost of not too much area, because a large k means small basic cells and there are not many redundant cells needed to enhance the yield significantly, as was already shown in a numerical example.

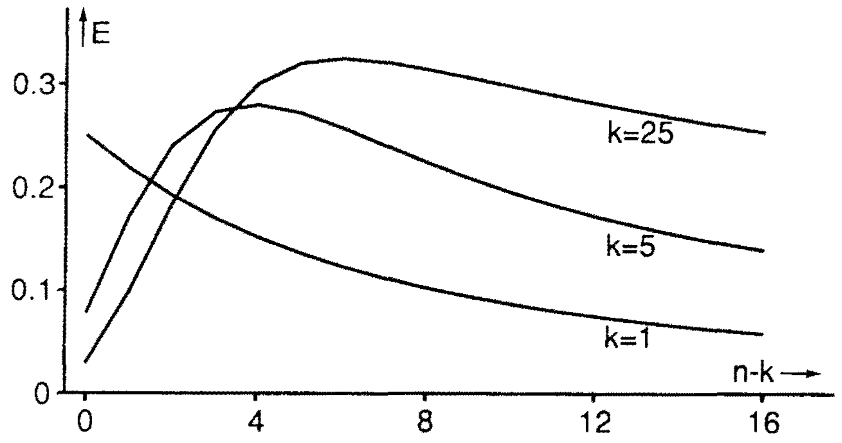

Figure 3. An example of the behaviour of the efficiency E as a function of the available number of cells (n), with the required number of cells (k) as a parameter.

Special attention should be given to the fact that there apparently exists a critical circuit-area a_{crit}, below which the partitioning of the circuit is undesirable, but above which a significant degree of partitioning is required to reach the maximum efficiency. This can be explained by looking at a graph of the maximum efficiency E_{max} as a function of the number k of required cells, for various values of a, as depicted in figure 4.

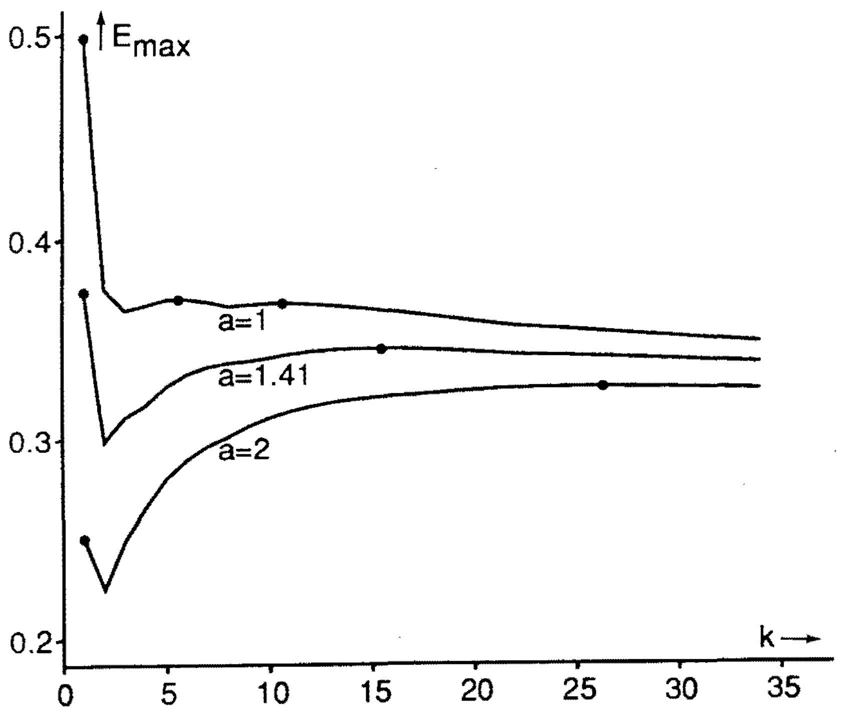

Figure 4. An example of the behaviour of the maximum efficiency E_{max} around $a = a_{crit}$, as a function of the required number of cells (k), with the desired circuit area (a) as a parameter.

As can be seen, the graphs of figure 4 may show more than one (local) maximum, indicated in the figure with dots. The heights of these maxima all depend on the circuit size a, but with different sensitivities. For small values of a, the maximum at $k = 1$ will be the only, or at least the highest one. At a given value of $a = a_{crit}$, another local maximum, at a significantly larger value of k, will be equally high as the maximum at $k = 1$. From this specific a_{crit} upwards, partitioning of the circuit and the application of redundancy will be favourable.

3.f. Influence of the technological parameters w and b.

Of course, all results as given in table 7 depend on the technological parameters w and b. As an illustration, table 8 gives the results calculated for a given $a = 8$ and for various values of w and b. As can be seen, the required total area as well as the maximum efficiency strongly depend on w, but not so much on b.

w	b	k	n	n/k	Y	A_t/A_c	E_{max}
0.1	1.0	193	211	1.09	0.9912	1.92	0.5154
0.1	1.5	174	190	1.09	0.9920	1.90	0.5208
0.1	2.0	164	179	1.09	0.9924	1.89	0.5238
0.3	1.0	183	213	1.16	0.9902	3.79	0.2614
0.3	1.5	135	158	1.17	0.9863	3.66	0.2698
0.3	2.0	117	137	1.17	0.9863	3.58	0.2752
0.5	1.0	234	277	1.18	0.9911	5.84	0.1697
0.5	1.5	167	199	1.19	0.9908	5.59	0.1772
0.5	2.0	130	156	1.20	0.9870	5.41	0.1823

Table 8. Illustration of the effect of the technological parameters w and b on partitioning, redundancy, area and maximum efficiency for a given desired circuit area $a = 8$.

Also, the critical circuit area a_{crit}, at which the maxima at $k = 1$ and at $k > 1$ in figure 4 have equal heights, depends more on w than on b, as is illustrated in table 9.

4. Application to specific architectures.

The model which was presented in the previous section, was put in general terms and was not based on any specific architecture. It was assumed that a circuit could be partitioned into any number of cells, that all these cells were identical and that the circuit could be reconfigurated in such a way that any faulty cell could be replaced by any good cell. Of course, this will generally not be true in practice. Nevertheless, the model can be used for a wide variety of architectures, provided the following adaptations are being made.

w	b	a_{crit}
0.1	1.0	0.71
0.1	1.5	0.70
0.1	2.0	0.69
0.3	1.0	1.62
0.3	1.5	1.57
0.3	2.0	1.52
0.5	1.0	2.27
0.5	1.5	2.19
0.5	2.0	2.13

Table 9. Illustration of the effect of the technological parameters w and b on the critical circuit area a_{crit}.

1) It is, of course, not always possible to partition a specific circuit into any number of sub-circuits. The micro-architecture will often cause a preference for certain numbers of cells, for instance powers of two. In practice, this means that not all natural numbers should be allowed as possible solutions of k, but that solutions of k should be restricted to a limited set of integer values fitting the micro-architecture of the specific circuit under development. This will not influence the previously mentioned results significantly, because the efficiency of the optimal partitioning will not show a very sharp maximum (see figure 4). This can, for instance, be illustrated by table 10, which shows the calculated results in the case of a = 8, around the maximum efficiency, which will occur at k = 135 (w = 0.3, b = 1.5, see table 7).

As can be seen from the above-mentioned and other experiments, a slight variation in k around maximum efficiency will not significantly influence the total area, efficiency and optimal ratio of redundancy.

k	n	n/k	Y	A_t/A_c	E_{max}
136	159	1.17	0.9862	3.65	0.26982729
135	**158**	**1.17**	**0.9863**	**3.66**	**0.26982730**
134	157	1.17	0.9863	3.66	0.26982559
↓	↓	↓	↓	↓	↓
128	151	1.18	0.9866	3.66	0.26977643

Table 10. The effect of small variations in k around the value of maximum efficiency (a = 8, w = 0.3, b = 1.5)

2) From table 7 it can be seen that a circuit with a size of, for instance, a = 32 has to be partitioned into 673 sub-circuits to give an optimal efficiency. In practical situations, this number may be considered rather large. Table 11 shows the effect of rounding k to the nearest lower and subsequently lower powers of two

(w = 0.3, b = 1.5). From this table it can be seen that, in this case, solutions down to, say, k = 128 still show an acceptable efficiency.

k	n	n/k	Y	A_t/A_c	E_{max}
673	763	1.13	0.9955	4.33	0.2300
512	601	1.17	0.9951	4.34	0.2291
256	342	1.34	0.9895	4.54	0.2178
128	215	1.68	0.9795	5.21	0.1881
64	160	2.50	0.9618	7.00	0.1374
32	158	4.94	0.9319	12.34	0.0755
16	250	15.63	0.8834	34.37	0.0257

Table 11. The effect of decreasing the rate of partitioning on required redundancy, area, yield and efficiency (a = 32, w = 0.3, b = 1.5)

3) So far, it was assumed that the desired circuit could be partitioned into any number of identical cells. This may be acceptable for very regular structures, like memories and systolic arrays, but not for all types of circuits. The model of this memorandum may therefore predominantly be useful for regularly structured architectures. Nevertheless, the architecture of almost any circuit allows a distinction of large functional blocks with a regular structure. Applying the model on each of these blocks separately, will also give a (nearly) optimal solution for the complete circuit.
Moreover, there are three reasons causing the area of a wafer-scale integrated circuit to be significantly larger than that of a VLSI-circuit. Firstly, a WSI-circuit will by nature incorporate a certain amount of redundant circuitry. Secondly, a relatively large area will be occupied by the implementation of the additional wiring and switches needed to (re-)configurate the circuit during the initial test or any power-up routine. Thirdly, the development of WSI-circuits will take some time, during which regular VLSI-circuits will continue to develop as well. In order to produce economically competitive WSI-circuits, one will have to aim at circuit sizes which are at least half an order of magnitude larger than state-of-the-art VLSI-circuits. The size of realistic WSI-circuits will therefore be of such a magnitude that they will inevitably *have* to be more or less regular in order to be designed by presently available CAD-tools.

4) In the model, it was assumed that any faulty cell could be replaced by any spare cell. In a practical situation, this will generally not be true. A faulty cell will have to be replaced by a cell in its close vicinity. Often, the implementation provides for complete rows and/or columns of spare cells, and reconfiguration is performed by implementing the proper "sidesteps". To the model, this means that n can not adopt any integer value $k \leq n < \infty$, but n will be restricted to certain architecture-dependent values. These restricted values of n can, just like the above-mentioned restricted values of k, be incorporated into the algorithm of the model, which will then find the optimal solution under the additional constraints. Moreover, all restrictions considering reconfiguration wiring and switches can be accounted for in the parameters w and b.

Conclusions and summary.

A model was presented which calculates the optimal degree of partitioning, rate of redundancy, total area, yield and efficiency for any desired (large) circuit. It was shown that there exists a critical circuit size, below which the implementation of partitioning and redundancy is undesirable. Above this critical circuit size, partitioning into rather small cells leads to optimal efficiency, in spite of the fact that small cells require a considerable overhead of reconfiguration wiring and switches. This also means that the implementation of very large redundant cells or blocks will cost more area than the enhancement of efficiency justifies. It was shown that the model can be adapted to any specific architecture by adjusting parameters and limiting possible solutions to restricted, architecture-dependent values. By a simple mathematical substitution, the model can be made completely analytic, making it very simple to investigate the sensitivity of the optimal solution to small variations in parameters.

Appendix.

In expression (18), the faculty functions may cause some difficulties, due to the rapid increase of n! with increasing n. In many applications, the numeric value of n! will exceed the range of the integer or even real variables on the used computing instrument. The calculated results, however, are in a reasonable range, as n! is divided by (n-i)!(i)! in all cases, due to the binomial origin of the faculty functions. Out-of-range-problems can therefore be avoided by calculating with the logarithms of the faculty functions.
On the other hand, these faculty functions are the only functions causing (18) to be non-analytic. In order to take the derivative of (18) with respect to its parameters, for sensitivity calculations, it may be found very useful to substitute n! by its well-known Stirling-approximation [e.g. 10,11].

$$n! \approx (n/e)^n . \sqrt{(2\pi n)} . C(n) \qquad (A.1)$$

This approximation is based on the theory of gamma-functions. Its derivation is beyond the scope of this memorandum. C(n) is a correction factor, consisting of a series expansion of n^{-1}

$$C(n) = 1 + (C_1/n) + (C_2/n^2) + (C_3/n^3) + (C_4/n^4) + \dots \qquad (A.2)$$

$C_1 \dots C_r$ are the results of first order Bernouilli-functions. In most publications, this correction factor is truncated after the fifth term ($C_r = 0$ for $r > 4$). The numeric values of $C_1 \dots C_4$ are

C_1	=	1/12	=	$8.3333333 \cdot 10^{-2}$
C_2	=	1/288	=	$3.4722222 \cdot 10^{-3}$
C_3	=	-139/51840	=	$-2.6813271 \cdot 10^{-3}$
C_4	=	-571/2488320	=	$-2.2947209 \cdot 10^{-4}$

Table A.1. Numeric values of the coefficients in the correction factor C(n).

Let SLA ("Stirling Low Accuracy") be the Stirling approximation of n! with C(n)=1 (meaning $C_r = 0$ for $r > 0$), let SMA ("Stirling Medium Accuracy") be the Stirling approximation of n! with $C(n) = 1 + (1/12n)$ (meaning $C_r = 0$ for $r > 1$), and let SHA ("Stirling High Accuracy") be the Stirling approximation of n! with C(n) truncated after the fifth term (meaning $C_r = 0$ for $r > 4$). Then, the accuracies as summarized in table A.2 for the approximations are obtained. For $n \rightarrow \infty$, n!/SLA, n!/SMA and n!/SHA all approach unity.

n	n!/SLA	n!/SMA	n!/SHA
0	∞	∞	∞
1	1.08444	1.001019	1.00050078
2	1.04221	1.000519	1.00002102
3	1.02806	1.000279	1.00000300
4	1.02101	1.000171	1.00000073
5	1.01678	1.000115	1.00000024
6	1.01397	1.000083	1.00000010

Table A.2. Accuracies of the Stirling approximation of n! with several numbers of terms in the correction factor C(n) taken into account.

Obviously, the SLA-approximation has the disadvantage of an unacceptably low accuracy for low values of n. The SHA-approximation on the other hand, provides an excellent accuracy (for $n > 0$), but will be cumbersome to differentiate, due to the relatively high number of terms in the correction factor C(n). In most applications, the SMA-approximation will show a useful balance between both.

The difficulty of the infinite error arising when calculating 0! can be overcome by recognizing that $n! = (n+1)!/(n+1)$. Combining this with (A.1) and (A.2) where $C_r = 0$ for $r > 1$ yields

$$n! \approx \sqrt{(2\pi/e^3)} \; . \; [(n+1)/e]^{n-1/2} \; . \; (n+{}^{13}/_{12}) \qquad \text{(A.3)}$$

This analytic approximation of n! shows an error ranging from 0.1% for n = 0 to less than 0.01% and approaching zero for n > 5.

References.

1. E.g. all publications on the Philips/Siemens Mega-project.
2. J. Middelhoek. "De juiste connecties". Proceedings of the Holland Elektronica Interconnectiedag, Holland Elektronica, Zoetermeer, The Netherlands, ISBN 9071.306-10-0, March 1988, pp. 43-49.
3. Gabrièle Saucier and Jacques Trilhe. Editorial introduction to Gabrièle Saucier and Jacques Trilhe (editors), Proceedings of the IFIP WG 10.5 Workshop on Wafer Scale Integration, North Holland, ISBN 0-444-70103-6, March 1986, pp. V-XX.
4. Jim C. Harden, Noel R. Strader II. "Architectural Yield Optimization for WSI". IEEE Transactions on Computers, vol. 37, no. 1, January 1988, pp. 88-110.
5. Hans-Jürgen M. Iden. "On the Optimization of Hierarchical Redundancies Including Configuration Nets and Switches". Earl Swartzlander and Joe Brewer (editors), Proceedings of the First International Conference on Wafer-Scale Integration, IEEE Society Press, ISBN 0-8186-9901-9, January 1989, pp. 173-182.
6. Tom Leighton, Charles E. Leiserson. "A Survey of Algorithms for Integrating Wafer-Scale Systolic Arrays". Gabrièle Saucier and Jacques Trilhe (editors), Proceedings of the IFIP WG 10.5 Workshop on Wafer Scale Integration, North Holland, ISBN 0-444-70103-6, March 1986, pp. 177-195.
7. François Blayo, Philippe Hurat. "A Reconfigurable WSI Neural Network". Earl Swartzlander and Joe Brewer (editors), Proceedings of the First International Conference on Wafer-Scale Integration, IEEE Society Press, ISBN 0-8186-9901-9, January 1989, pp. 141-150.
8. R.M. Lea. "A WSI Image Processing Module". Gabrièle Saucier and Jacques Trilhe (editors), Proceedings of the IFIP WG 10.5 Workshop on Wafer Scale Integration, North Holland, ISBN 0-444-70103-6, March 1986, pp. 43-58.
9. Ben Warren, Wayne Richardson, Keigi Kanegawa, Cliff Arnell. "A One Megabit SRAM Fabricated with 1.2 µm Technology". Earl Swartzlander and Joe Brewer (editors), Proceedings of the First International Conference on Wafer-Scale Integration, IEEE Society Press, ISBN 0-8186-9901-9, January 1989, pp. 47-53.
10. I.N. Sneddon (editor). "Encyclopaedic Dictionary of Mathematics for Engineers and Applied Scientists". Pergamon Press, Oxford, ISBN 0-08-016767-5, 1976, First edition, p. 641.
11. Granino A. Korn, Theresa M. Korn. "Mathematical Handbook for Scientists an Engineers". McGraw-Hill Book Company, New York, 1961, pp. 698-699.

Efficient Implementation of High-Level Parallel Symbolic Languages

Mark Korsloot[2]
Dept. of Electrical Engineering
Delft University of Technology
PO.Box 5031, 2600 GA Delft, the Netherlands
mark@duteca.et.tudelft.nl

Abstract

A compile-time technique is presented for determining if a set of procedures within a parallel program can be executed sequentially without causing deadlock. The analysis and methods are described for committed-choice parallel logic programming languages; however, the concepts are general enough for any concurrent languages with fine-grain communicating processes. We derive methods for ensuring that sequential evaluation of a program module cannot result in producer-consumer suspension within the module itself, thereby resulting in deadlock. The advantages of sequentializing fine-grain languages include the use of "traditional" compiler optimizations, such as global register allocation, and continuation-stacking procedure invocation.

1 Introduction

Traditional parallel procedural languages evolved from sequential programming languages. The quest to uncover more parallelism, in more efficient ways, is paramount in the development of these languages. Alternatively, concurrent languages, such as committed-choice parallel logic programming languages, have a great deal of inherent parallelism. These languages and their implementations have been refined to exploit the parallelism more and more efficiently, with increasingly sophisticated interpreters and emulators, in both software, firmware, and hardware. Less research has been done concerning efficient compilation for the parallel execution of these concurrent languages.

In this paper, we introduce a method to safely sequentialize pieces of concurrent programs, with the intention of increasing execution speed. Specifically, we describe our method with respect to the family of flat committed-choice parallel logic programming languages, such as FCP, FGHC, and Parlog [16]. To achieve serialization, we combine a general mode-analysis algorithm [20] with a goal-ordering algorithm described here. We fully describe the conditions under which the analysis can successfully sequentialize a program, and indicate practical uses of the technique, such as global register allocation and continuation-based goal management.

There are several reasons why sequential execution can be beneficial. One of the advantages of sequential code blocks is to increase granularity [5, 18]. An often-mentioned problem of committed-choice languages is their small average granularity, causing a high overhead for process management and an abundance of light-weight processes. By sequentializing portions of a fine-grain parallel program, execution time decreases and processor utilization increases. Apart from this, sequentializing part of a program also makes it possible to use many well-known compiler optimization techniques (e.g., [1]), such as a global or interprocedural register

[2]This work has been done in cooperation with Prof. E. Tick, Dept. of Computer Science, University of Oregon

allocation scheme. The trick is to determine which sets of procedures should be serialized for overall benefit, and if they can be serialized safely, i.e., without chance of deadlock.

Our ultimate goal is to exploit efficient memory management, optimal register allocation, and destructive variable assignment techniques within "sequential" program modules. Thus we can compile parts of committed-choice programs as if they were sequential, procedural languages, thereby increasing the average granularity. By judicious modularization of a program, overall efficiency of committed-choice programs will improve significantly with relatively simple analysis.

This paper is organized as follows. A brief review of committed-choice languages and argument modes is given in Section 2. Section 3 shortly describes a first attempt at efficiently implementing committed-choice languages by exploiting directional programs, as first discussed by Gregory [8]. Section 4 describes a more general method of mode analysis, due to Ueda [20]. The results of this mode analysis can be used to ensure safe sequential execution, as described in Section 5. Section 6 describes how the preceding methods can serve as the basis for the efficient implementation of sequential modules, and Section 7 summarizes the paper.

2 Background and Terminology

In this section a brief review of committed-choice languages and argument modes is given. The terminology introduced here, for *input and output modes*, *modules*, *basic blocks*, and *internal and external suspension*, is necessary for the remainder of the paper. In the literature, the notion of modes is somewhat overloaded in its meaning, through sloppy usage and intuitive bias. We hope to clarify the intended meanings. It is important to stress that we limit ourselves to the family of flat committed-choice languages [16]. This limitation is not a severe handicap, as the expressive power of these languages is comparable[3] to those languages with deep guards [17]; however, their implementation is simpler and more streamlined.

In flat committed-choice languages, Horn clauses have the form

$$H :- G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_n.$$

where m and n are zero or positive integers. H is the clause head, G_i is a guard goal, and B_i is a body goal. A *goal* can be considered a procedure invocation. A *conjunction* of goals is simply a set of goals, appearing within the same clause. The commit operator '|' divides the clause into a passive part (the *guard*) and active part (the *body*). For flat languages, the guard goals can only be built-in predicates, such as `integer(X)`, or `X>0`.

Both the clause head and internal goals can have arguments. The head arguments correspond to formal parameters in a procedure definition, whereas the guard and body goal arguments correspond to passed parameters in a procedure call. A *procedure* is defined as the set of clauses having the same name and number of head arguments.

An important concept is the notion of *modes*. A procedure argument is used to communicate values between the caller and callee. Intuitively, when the caller passes a value into the callee, the corresponding argument is used in an input mode. Similarly, when the callee passes a value back to the caller, the argument is used in an output mode. However, due to the nature of the logical variable, a single argument, in a given invocation, can be used both to pass values in and return values back. One example of this is called "incomplete messages," e.g., `m(X)` is an input message containing an unbound variable `X`, meant to be bound with a return message by the callee.

[3]See Shapiro [15] for an in-depth comparison of this family of languages.

Mode declarations can be either explicit (as in Parlog [8]), or implicit. A mode can be either *input* or *output*. In Parlog, the user must declare the modes for each "top-level" argument of a predicate. A top-level argument is the outermost term passed to a procedure, distinct from any subterms that it may be composed of. The following definitions are due to Gregory [8] and others.

> *Definition*: An *input argument* of a goal, denoted here by mode '?', is an argument which is either instantiated when the goal is called, or it is a variable. However, during head matching or guard evaluation, when the input argument is a variable, it can never be instantiated by the head-matching process. Basically, only input matching is performed. □

> *Definition*: An *output argument*, denoted here by mode '^', is an argument which is always unbound in the caller, and whose value is bound by the callee. □

In the following, arguments of a predicate which have been declared as input or output will be named *input positions* or *output positions*, respectively. Furthermore, the goal(s) which instantiate a variable are called the *producer(s)* of this variable, while the goal(s) which use, but not instantiate, a variable are called the *consumer(s)* of that variable.

There are some flaws, however, in the simple mode system. Primarily, the modes concern only top-level functors. For example, consider the following program:

```
mode f(?,^).
f([X|Xs], Z) :- true | g(X),h(Xs,Z).
```

The mode declaration of the first argument does not convey anything about the value of `X` when `f/2` is called: it only specifies that the first argument of the caller must be a (non-empty) list. Furthermore, specifying an argument as input does not exclude the possibility that the argument is bound to a non-variable term in the body, e.g.,

```
mode f(?).
f(X) :- true | X=[].
```

The above is a legal Parlog program, and when called with the query "`?- f(X)`," `X` will be bound to `[]`, which contradicts the intuitive notion of input argument.

The notions of input and output modes, as introduced by Ueda [20], are somewhat more consistent in this regard. Instead of simply specifying the top-level arguments, Ueda's method attempts to infer the modes of *all* variables and structures occurring in the clause. Ueda distinguishes terms by specifying the *path* which must be taken to get to a specific term. For example, the path to get to variable `X` in predicate `f/2` above is specified as `<f,1><.,1>`,[4] i.e., the head of the first argument of `f/2`.

The *value* of a path is then defined as the first (principal) functor following this path when the term at the end of the path is instantiated. In our previous example, if `X` is instantiated to `a(0,0)`, then the value of the path `<f,1><.,1>` is `a/2`. We now give definitions of input and output path modes due to Ueda [19].

> *Definition*: If a path is defined as *input*, the value (i.e., the principal functor) of this path *may* (but need not) be bound by the caller, and *will never* be bound by the callee. □

[4]For lists, the functor `./2` is used, so `<.,1>` is the head of a list, while `<.,2>` is the tail of a list.

Definition: If a path is defined as *output*, the value (i.e., the principal functor) of this path *will never* be bound by the caller, and *may* (but need not) be bound by the callee. □

We now define the notions of a *sequential module* and a *sequential basic block*. We use the term *module* to describe a set of self-contained predicates, which is entered (called) through a single entry-point (the *module entry-point*). In this context, "self-contained" means that no predicate inside the module calls any predicates outside the module, nor does any predicate outside the module call any predicate inside the module, apart from the visible module entry-point. For example, consider the following code segment:

```
a :- b,c,d.
b :- e,b.
c :- b.
d.
e.
```

Predicates b and e comprise a module; this module is entered only through b. From within this module, there are no calls to predicates outside it. Extending this notion, a complete program can be viewed as a module with the top-level query calling the module entry-point (assuming that the query consists of a single goal only). If the goals inside a module are executed sequentially, the module is called a *sequential module*, for obvious reasons.

A *basic block* is defined simply as a group of body goals within the same clause. Note that this definition is more general than the standard definition for machine instructions [1], although the intention is similar. If these body goals are executed sequentially, the block is called a *sequential basic block*. Grouping together these body goals can be done in several ways. First, the goals can be folded together (either by the user or as a source-level transformation by the compiler) into another goal, which is specifically marked as "sequential." For example, in the previous code, if we group the goals c and d, the program could be transformed into:

```
:- sequential g.
a :- b,g.
b :- e,b.
c :- b.
d.
e.
g :- c,d.
```

A second possibility for grouping a set of body goals is using an operator such as the sequential conjunction operator '&' from Parlog:

```
a :- b, (c & d).
```

In the third option for marking a sequential basic block, no explicit transformations or annotations are made, but rather the compiler detects the presence of a sequential basic block, and implicitly transforms the block into an internal representation.

Extending these notions with respect to the call graph of a program, a module is defined as a subgraph of the entire graph. This subgraph is entered through only one node,[5] and has no edges to nodes outside the subgraph. In contrast, a basic block is a set of nodes which are all connected to the same parent. Recall that this implies that all goals in a clause can be executed concurrently. An example is shown in Figure 8a, which represents the aforementioned program. The left box represents a module, and the right box represents a basic block.

[5]There may be multiple edges entering a module, but they can be connected only to the module entry-point.

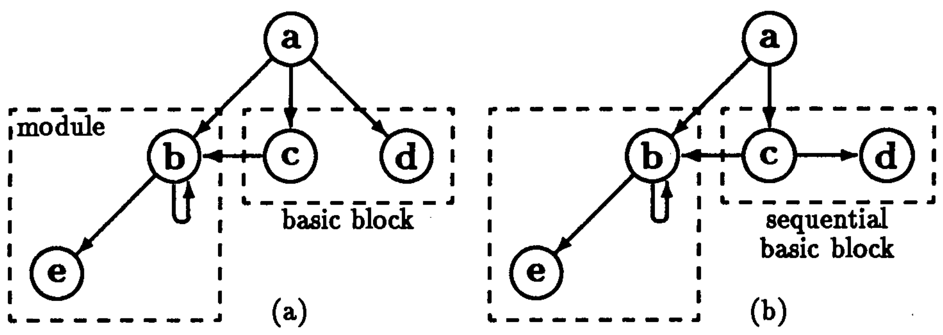

Figure 8: (a) Modules and Basic Blocks in Call Graphs. (b) A Call Graph with a Sequentialized Basic Block.

By sequentializing a basic block, the call graph changes as follows: if a basic block consists of n goals, for all but the first goal, the edges connecting them to their parent are removed from the call graph, and the n goals are connected such that goal i is connected to $i+1$ (for all $i < n$). This is shown in Figure 8b.

Algorithms for *selecting* sequential modules and basic blocks (i.e., their composite procedures and appropriate size) are an ongoing research topic of great importance, related to granularity analysis (e.g., [5, 18]). However, this is beyond the scope of our paper.

Another important notion to be defined in this context is that of *suspension*. The execution of a committed-choice goal will *suspend* whenever an input variable is not sufficiently instantiated for the goal to commit. This situation changes when another process further instantiates the variable, allowing the suspended process to resume. If the variable is never instantiated, *deadlock* will arise, which is defined as a situation where one or more suspended processes exist, but no runnable processes exist. We define suspension with respect to both modules and basic blocks:

> *Definition*: A process (or a set of processes) is suspended *externally*, when the cause of the suspension lies outside the suspended module (basic block). □

For example, consider the query "`?- c(X),p(X).`" where `c/1` (the consumer of `X`) and `p/1` (the producer of `X`) are different modules. If `c/1` is executed first, then external suspension of `c/1` occurs. Intuitively, external suspension means that a process, external to the module, which produces data for the module, has run of out of data, and the module must wait for this producer to produce more data. When a process is suspended on more than one variable, with at least one of the causes of suspension outside the module, this is also called external suspension.

> *Definition*: A process (or a set of processes) is suspended *internally*, when the cause of the suspension lies inside the suspended module (basic block). □

Intuitively, internal suspension means that some variable which is local to the module or basic block is not sufficiently instantiated to allow the process to continue. If a process is suspended on more than one variable, suspension is called internal if and only if all causes are inside the suspended module. With these definitions, the following theorem is easy to derive:

> **Theorem 1** *If the goals inside a module or basic block are executed sequentially, then internal suspension will always cause deadlock for the top-level call of the module entry-point.*

The proof is simple: if suspension occurs, the module has to wait for some other process to produce more data. However, this process is located inside the module, and will never be executed because the goals inside the module are executed sequentially. Therefore the call to this module deadlocks. Although this theorem may be obvious, its importance is that it clearly shows that internal suspension must be prevented for sequential basic blocks and modules. In the next section we begin to develop conditions under which this is true.

3 Directional Programs

One of the first attempts towards a parallel logic language which could be implemented efficiently on a distributed, loosely-coupled architecture was the Relational Language [3], the direct precursor of Parlog [8]. Apart from the mode system, which is also present in Parlog, the Relational Language featured *strong* arguments.

> Definition: A *strong argument* of a procedure invocation, if it is in an output position, is a term that is completely constructed by a single body goal in that procedure's definition, without any contribution from other goals in this conjunction. The constructing goal is often called the *producer*. If a strong argument is in an input position, it is completely constructed by external goal(s), and no bindings to it are made in this procedure invocation. □

Thus there can never be an output substitution for any variable occurring in a strong input argument position. This does not preclude the construction of terms containing unbound variables, but these variables can *never* be instantiated within the conjunction by a goal other than the producer. As an example, consider the following code segment:

```
mode f(^).
f1(X) :- X = g(0).
f2(X) :- X = g(Y).
f3(X) :- X = g(Y), Y = 0.
```

The `X` arguments in both `f1/1` and `f2/1` are strong (their values are completely constructed by a single body goal), whereas the `X` argument in `f3/1` is not strong. Note that `Y` is not instantiated by another body goal in `f2/1`. The implication of strong variables is far reaching: "back communication" (e.g., the use of incomplete messages) is impossible. In general, the full power of the logical variable is lost.

For completeness, we now discuss the status of variables that are not strong. To retain the full power of the logical variable, *weak arguments* are necessary. For example, argument `X` in `f3/1` is *weak* because, its value is constructed by two body goals within the same clause. For output arguments, the distinction between weak and strong is not important (in both cases, output bindings are only made through variables occurring in output positions), therefore in the following we will distinguish only between strong and weak *input* arguments.

> Definition: A *weak input argument* of a goal is one in which variables might be instantiated by the evaluation of the goal. □

Weak input arguments were introduced in Parlog (and implicitly used in other committed-choice languages), because Parlog programs only specify the modes for top-level arguments, without considering subterms of these top-level arguments. Consider, for example, the following correct Parlog program:

```
mode f(?).
f(g(X)) :- X=[].
```

Although the weak argument of `f/1` is (correctly) defined as input, because it is a structured term `g/1`, the variable `X` inside this argument will be instantiated when `f/1` is called.

With the definition of strong arguments, a *directional program* can be defined:

> Definition: A *directional logic program* is a program in which all arguments (of all clauses of all procedures) are strong. □

A directional module can be defined similarly. In a directional program, the mode declarations indicate which body goal constructs the value of a variable: the goal in which the variable appears in an output argument position. Gregory [8] uses directionality only to check for the compile-time safety of guards, something which we do not discuss here. However, the directionality mechanism is more powerful — we show, in Theorem 2, that it can be used to significantly lower the number of suspensions:

> **Theorem 2** *If a program is directional, and given a correct ordering of the body goals, no deadlock will occur, and the program can only suspend externally, i.e., on variables whose producers are the top-level query.*

Again, this theorem can easily be extended to cover modules. For example, consider the body goals "`f(A), g(A,B), h(B).`" Suppose `f/1` is the producer for `A` and `h/1` is the producer for `B`. A correct ordering, to avoid deadlock during sequential execution, is "`f(A) & h(B) & g(A,B).`"

It may be the case that more than one ordering exists satisfying this theorem; however, it is also possible that no such ordering exists, thus leading to internal suspension of a directional program. For example, consider the body goals "`f(A,B), g(A,B)`" where `f/2` is producer for `A` and `g/2` is producer for `B`. In this case, the program is still directional, but there is no ordering of `f/2` and `g/2` satisfying Theorem 2.

When body goals are executed in the "correct order," the number of suspensions is reduced because a producer of a variable is always executed before its consumer(s). Although we will not go into further details here, [12] illustrates why the requirement for a program to be directional (i.e., for all arguments to be strong) is too restrictive, and in the remainder of this paper we will describe an approach to ease this requirement, while retaining the full use of Theorem 2.

4 Mode Analysis

As the requirement for full directionality with only strong arguments is too restrictive to be effective, another type of mode analysis is necessary. A viable alternative is the mode analysis described by Ueda [20] (other options, which we do not consider in this paper, are available, such as abstract interpretation [13]). Ueda's mode analysis automatically infers the modes of all arguments at the top-level *and* those inside structured top-level arguments. A mode is either *input* or *output*[6] as defined in Section 2.

4.1 Simplified Rules

The mode analysis described in this section reviews Ueda's technique, although a different (hopefully more clear) exposition and less formal notation is used here. The mode analysis uses the notion of a *path* to denote a specific (textual) occurrence of a term. A path describes how the different layers of structured data elements must be "peeled off" to get to this specific occurrence. For example, given the clause head `f(X,g([X|_]))`, the path to the first occurrence

[6] Shortened to "in" and "out" in certain contexts.

of `X` is `<f,1>`, while the path to the second occurrence is `<f,2><g,1><.,1>`. This second path means "take the second argument of `f`," then "take the first argument of `g`," and then "take the first argument of the list."

The mode of a path p, which is denoted as $m(p)$, is defined as either input or output. The definitions of input and output in [20] are somewhat ambiguous, so we use two improved definitions [19]. Let p be a path leading to a variable. Then:

- "$m(p)$ = in" means that the value (i.e., the principal functor) of the variable at p *may* (but need not) be bound by the caller, and *will never* be bound by the callee.
- "$m(p)$ = out" means that the value (i.e., the principal functor) of the variable at p *may* (but need not) be bound by the callee, and *will never* be bound by the caller.

Intuitively, this implies that an input path will not be (further) instantiated, while an output path implies that a process can never suspend on the value of the path. Note that variable-variable unification does not count as "binding the value" of a variable. Thus, input paths leading to variables cannot cause suspension, even if the incoming argument is unbound.

If possible, the modes of all paths must be inferred to find a safe goal ordering, which avoids internal suspension. The mode of a path can be found by applying the following rules:

§1. For some path p in a clause, $m(p)$ = in, if either

1. p leads to a non-variable in the head or body, or
2. p leads to a variable which occurs more than once in the head, or
3. p leads to a variable which also occurs in the guard at path p_h and $m(p_h)$ = in

§2. Two arguments of a unification body goal (`=/2`) have opposite modes, for all possible p, or more formally: $\{\forall p \; \mathtt{m}(<=,1>\mathtt{p}) \neq \mathtt{m}(<=,2>\mathtt{p})\}$.

To better understand the modes, we now give the intuition for these rules. If a path leads to a non-variable in the clause, then the value of the path is already known, and it will not be instantiated by the callee, thus its mode is 'in.' If a variable occurs more than once in the head, it can only be used for equality checking before commitment.[7] No bindings are allowed, thus again its mode is 'in'. Similarly, if a variable in the guard is used for checking (i.e., its mode is 'in'), then it is clear that value of the path to that variable is provided by the caller, so its mode is 'in.'

Because of the nature of unification, one of the arguments of `=/2` will function as producer for a specific path, while the other argument functions as consumer. Note, however, that §2 does *not* require that for all possible paths p, the modes of an argument are the same. For example, given the unification `[1,X]=[Y,2]`, `m(<=,1><.,1>) = in`, while `m(<=,1><.,2>) = out`.

When a variable occurs more than once, with at least one occurrence in the body, the situation gets more complicated. Because paths to variables occurring more than once in the head all have the same (input) mode (see §1b), it is correct (and simpler) to count only one occurrence of a variable in the head in the following rules.

§3. If there are exactly two occurrences, we have two possibilities:

[7]Note that these semantics are particular to FGHC. For example, the procedure "`f(X,X) :- X=3.`" can only succeed with the query "`?- f(3,3).`" If the intention was to output two copies of 3, then the proper code is: "`f(X,Y) :- X=3, Y=X.`" Parlog avoids this problem by allowing both arguments to be explicitly defined as output modes.

1. If both occurrences are in the body, the modes of their paths are inverted.
2. If there is one occurrence in the head and one in the body, the modes of their paths are the same.

§4. If there are more than two occurrences of a shared variable (i.e., at least two occurrences in the body), the situation is even more complex:

1. If the body contains more than two occurrences of the shared variable and the head has no occurrences, then one of the modes is 'out,' and the others are 'in.'[8]
2. If the head contains one occurrence of the shared variable (so the body has two or more occurrences), then the modes are as follows:
   (a) The mode of the head occurrence is 'in' iff the modes of all body occurrences are 'in.'[9]
   (b) The mode of the head occurrence is 'out' iff *one* of the body occurrences is 'out,' and the other body occurrences are 'in.'

The inversion in §3 can be intuitively explained by looking at an input path in a clause head. For that clause, the variable acts as a consumer of data (therefore its mode is input). However, within the clause the variable in the head acts as a producer for the body of the clause, thus inverting its mode within the clause. The opposite of this holds for an output path in the head.

The complexity of §4 can be explained intuitively by looking at the inversion of the modes discussed previously, combined with the fact that only one occurrence of a variable can be its actual producer. The problem with this last rule is that it causes non-binary constraints to occur. To clarify this, we will give two examples of how the modes of a shared variable can look when multiple (three in this case) occurrences are present.

f(X)	:-	b1(X),	b2(X).
in		in	in
out		out	in
out		in	out

f	:-	b1(X),	b2(X),	b3(X)
		out	in	in
		in	out	in
		in	in	out

4.2 An Example of Mode Analysis

To explain the concept of modes and paths, and to understand the rules given above, it is best give an example. Figures 9 and 10 give the mode analysis for quicksort. In these proofs, let q_i(p) ≡ m(<q,i>p) and s_i(p) ≡ m(<s,i>p), while '$=_k$' represents the k^{th} instance of the unification goal =/2, and 'ϵ' represents an empty path. Furthermore, each step in the proof is annotated with the rule used. Each mode relationship proved is called an *axiom*, e.g., there are six axioms comprising the full mode definition for q/3.

As an example of how such a proofs are constructed, consider the fourth axiom in Figure 10. We start with the path to the term on the right-hand side of $=_1$/2. The value of this path is a non-variable ($\Box$), so according to §1a, its mode is 'in.' Using §2, we then derive that the mode of the corresponding path on the left-hand side of $=_1$/2 is 'out.' As there are two occurrences of the variable S, one in the head and one in the body, §3b applies. If we now substitute p=ϵ in the last axiom derived, and combine it with the previous one, the final outcome is s3(ϵ) = out. The other axioms are proved similarly. Together these results give an idea how data flows

[8]This means that one of the occurrences is designated as the producer of this variable.

[9]Note that if a variable occurs more than once in the head, its mode is 'in' by §1b, implying that §4b-ii cannot be used.

q([], R0,R) :- R0 $=_1$ R.
q([X|L],R0,R) :- s(L,X,L1,L2),q(L1,R0,[X|R1]),q(L2,R1,R).

1.	q1(ϵ) = in	§1a
2.	q3(ϵ) = in	§1a, 2nd body goal
3.	q2(ϵ) = out	
a.	$\forall p$ q2(p) = m(<$=_1$,1>p)	§3b
b.	$\forall p$ q3(p) = m(<$=_1$,2>p)	§3b
c.	$\forall p$ q2(p) $\neq$ q3(p)	§2
d.	q2(ϵ) = out	"2"+c: sub p=ϵ
4.	$\forall p$ q1(<.,2>p) = s1(p)	§3b (on L)
5.	$\forall p$ q3(p) = q3(<.,2>p)	
a.	$\forall p$ q3(<.,2>p) $\neq$ q2(p)	§3a (on R1)
b.	$\forall p$ q3(p) = q3(<.,2>p)	"3.c" + a
6.	Three possibilities:	§4b
a.	q1(<.,1>p) = in, s2(p) = in, q3(<.,1>p) = in	
b.	q1(<.,1>p) = out, s2(p) = out, q3(<.,1>p) = in	
c.	q1(<.,1>p) = out, s2(p) = in, q3(<.,1>p) = out	
7.	s3(ϵ) = out	
a.	$\forall p$ q1(p) $\neq$ s3(p)	§3a (on L1)
b.	s3(ϵ) = out	"1" + a
8.	s4(ϵ) = out	see "7"

Figure 9: Mode Analysis Proof for Quicksort: q/3.

s(□, _,S,L) :- S $=_1$ □, L $=_2$ □.
s([X|Xs],A,S,L) :- A>X | L $=_3$ [X|L1], s(Xs,A,S,L1).
s([X|Xs],A,S,L) :- A=<X | S $=_4$ [X|S1], s(Xs,A,S1,L).

1.		s1(ϵ) = in	§1a
2.		s1(<.,1>) = in	§1c
3.		s2(ϵ) = in	§1c
4.		s3(ϵ) = out	
	a.	m(<$=_1$,2>) = in	§1a
	b.	m(<$=_1$,1>) = out	§2
	c.	$\forall p$ m($=_1$,1>p) = s3(p)	§3b
	d.	s3(ϵ) = out	c: sub p=ϵ
5.		s4(ϵ) = out	see "4"
6.		$\forall p$ s1(p) = s1(<.,2>p)	§3b
7.		$\forall p$ s4(p) = s4(<.,2>p)	
	a.	$\forall p$ s4(p) $\neq$ m(<$=_3$,2><.,2>p)	§3a (on L1)
	b.	$\forall p$ s4(p) = m(<$=_3$,1>p)	§3b (on L)
	c.	$\forall p$ s4(<.,2>p) = m(<$=_3$,1><.,2>p)	b: sub p=<.,2>
	d.	m(<$=_3$,1>p) $\neq$ m(<$=_3$,2>p)	§2
	e.	$\forall p$ s4(p) = s4(<.,2>p)	a+c+d
8.		$\forall p$ s3(p) = s3(<.,2>p)	see "7"
9.		s4(<.,1>) = out	
	a.	s1(<.,1>) = in	see "2"
	b.	$\forall p$ s1(<.,1>p) = m(<$=_3$,2><.,1>p)	§3b
	c.	m(<$=_3$,2><.,1>) = in	b: sub p=ϵ
	d.	m(<$=_3$,1><.,1>) = out	§2
	e.	s4(<.,1>) = out	see "7.b"
10.		s3(<.,1>) = out	see "9"

Figure 10: Mode Analysis Proof for Quicksort: s/4.

within this procedure, i.e., which arguments (and which variables within structures) are input, and which are output. This information can be used by a compiler to make optimal use of its resources, such as memory and registers.

To show that mode analysis can fail as well, consider the following example:

```
f :- r(X,Y), g(X), g(Y).
r(X,Y) :- X=Y.
```

Applying §3a to the first clause gives: $\forall p\ \mathtt{m}(<\mathtt{r},1>\mathtt{p}) \neq \mathtt{m}(<\mathtt{g},1>\mathtt{p})$ and $\forall p\ \mathtt{m}(<\mathtt{r},2>\mathtt{p}) \neq \mathtt{m}(<\mathtt{g},1>\mathtt{p})$. In other words: $\forall p\ \mathtt{m}(<\mathtt{r},1>\mathtt{p}) = \mathtt{m}(<\mathtt{r},2>\mathtt{p})$ However, applying §2 to the second clause gives: $\forall p\ \mathtt{m}(<\mathtt{r},1>\mathtt{p}) \neq \mathtt{m}(<\mathtt{r},2>\mathtt{p})$, which clearly contradicts the previous conclusion, so the mode analysis fails and no consistent modes can be found. This implies that this code cannot safely be considered as a sequential module, and standard compilation techniques must be used. The reason why the mode analysis fails can be seen more clearly from the unfolded program:

```
f :- g(X), g(X).
```

If both occurrences of X had mode 'in,' then there would be no producer for X. On the other hand, if both occurrences of X are 'out,' then they are competing to produce a value for X, but it is not certain which of the processes will generate the value. Therefore we cannot say which occurrence must be executed first to obtain a safe, i.e., deadlock-free, sequential module.

To come back, however, to the quicksort example, the most interesting point about this analysis is the sixth axiom in Figure 9. The analysis shows that three distinct modes of execution are possible for q/3, hinging on the use of shared variable X. Since X appears twice in the body of the second clause, the proof uses rule §4b. The three distinct modes depend on whether X is input by the clause head (first choice) or bound by a body goal (latter two choices). Given the mode analysis of s/4 in Figure 10, the second choice (b) is contradicted, since $\mathtt{s2}(\epsilon)$ = in. Therefore only two choices remain.

The problem is that these two choices are both valid: essentially Ueda's analysis indicates that without further information (e.g., modes of the query), X may be generated by the third argument of q/3 and output through the first argument (c), or vice versa (a). To the programmer, whose intent may have been the standard use of quicksort (a), this result may seem odd. However, choice (c) is valid — consider the query:

```
?- q([X],[3,2,1],[2,1]).
   X = 3
```

Although this is an artificially-created query, which is unlikely to appear in any program, this query is valid, and should execute successfully!

This example shows that to safely execute this module sequentially, more information is needed about the mode of the first argument. One way of finding this mode is by looking at the place(s) where the module is called. For example, given the definition and the modes of gen/2 in Figure 11, with $g_i(\mathtt{p}) \equiv \mathtt{m}(<\mathtt{gen},\mathtt{i}>\mathtt{p})$, look at the following call to the module:

```
...,gen(Max,List),q(List,SortedList,[]),...
```

Because only one occurrence of a variable in the body (or query) can have mode out and Figure 11 shows that $g_2(<.,1>)$ = out, we can conclude that $q_1(<.,1>)$ = in, which leaves us with only the normal use of quicksort (a).

As said before, one way of finding the extra information, needed to show that a module can safely be executed sequentially, is by using the context information, i.e., looking where the module is called. A second method is having the user explicitly specify modes to disambiguate

```
gen(0,X) :- G =1 [].
gen(I,X) :- I1:=I-1, X =2 [I|Xs], gen(I1,Xs).
```

1.	g1(ε) = in	§1a
2.	g2(ε) = out	§1a, §2, §3b
3.	g2(<.,1>) = out	"1", §4b-i, §2
4.	∀p g2(p) = g2(<.,2>p)	§3a, §3b, §2

Figure 11: Mode Analysis for **gen/2**.

such cases. For example, to indicate that an argument will be input throughout, i.e., for all subterms, the user can specify a *strong input argument*, perhaps with the mnemonic '??'. Note that this is not the same as fully ground!

One can even envision having the compiler query the programmer to provide this disambiguating information. The advantage of compiler queries is that only a small percentage of modes will likely need to be disambiguated. Thus the programmer would be saved from having to map out all the modes *a priori*, and need only specify ones that the analysis proves are ambiguous. We might add that this tends to be a good programming practice anyway! Good logic programmers specify the top-level argument modes of each procedure (possibly as a comment), but as shown, this is not sufficient documentation in all cases. For the example discussed above, the determination of safe sequential execution hinges on whether or not the head of the first argument is bound, so help is needed for this one mode only.

5 Goal Ordering

In this section we discuss how the mode information is used to reorder body goals within a clause to ensure safe sequential execution. For this, we have to introduce a *directionality rule*:

§5. For a variable occurring more than once in the body, the first "executed" occurrence[10] must be in an output position, thus later occurrences can only be in input positions.

This rule ensures that internal suspension is impossible in a sequential module. The basic idea is to find an ordering that does not contradict this rule, using the axioms derived by the mode analysis.

Now, by extending Theorem 2 to cover modules, and defining "correct ordering" to be that ordering of body goals which adheres to rule §5, the following theorem follows:

Theorem 3 *If, at compile-time, a consistent ordering for a sequential module can be found, then only external suspension can occur with respect to that module.*

In the rest of the program, other forms of suspension can occur, but the important point is that there can never be *internal* suspension in that module. If a consistent order can be found, the module can be compiled efficiently, as will be discussed in Section 6. Otherwise, compilation

[10]Assuming a left-to-right execution order, this is also the textually first occurring body goal.

of the sequential module defaults to standard committed-choice compilation techniques (e.g., [4, 10, 16, 17]).

We now present an algorithm for finding a consistent ordering. The first point to consider is that not all the mode axioms that fully describe a procedure are needed here, nor do we need the axioms in their full generality. Certain axioms, which we call *recursive axioms*, have a general form similar to axiom 5 of `q/3`: $\forall p$ `q3(p) = q3(<.,2>p)`. The meaning is that the mode of any third argument path is the same as the mode of the tail of that path. Intuitively, the procedure is recursing on the third argument, which is a list.

For the purposes of goal ordering, only the modes of variables appearing (syntactically) in the procedure definition are needed. Therefore all general paths p in the derived axioms can be instantiated to ϵ. For example, in the previous axiom, `q3(`ϵ`) = q3(<.,2>)`. Combining this with axiom 2 in Figure 9 allows us to derive that `q3(<.,2>) = in`. We call this process of instantiating general mode axioms into less general axioms, *recursive grounding.* As another example, combining axiom 4 in Figure 9 with axiom 1 in Figure 10 gives `q1(<.,2>) = in`.

After recursive grounding, we are left with two possible sets of modes for procedure `q/3`:

```
q([X|L],R0,R) :- s(L,X,L1,L2), q(L1,R0,[X|R1]), q(L2,R1,R).
q([?|?], ^,?) :- s(?,?, ^, ^), q( ?, ^,[?| ?]), q( ?, ^,?).
q([^|?], ^,?) :- s(?,?, ^, ^), q( ?, ^,[^| ?]), q( ?, ^,?).
```

The next step is to attempt to order the body goals in the second clause[11] so that the constraints implied by both sets of axioms (with respect to rule §5) are satisfied.

For each set of modes, the ordering algorithm iterates through each variable occurring more than once in the body: `X`, `L1`, `L2`, and `R1`. For each, constraint(s) are created relating the goals containing that variable. For example, for the first set of modes, `X` induces no constraints (since both the first and second goals use `X` for input). Examination of `L1`, however, induces the constraint: $G1 < G2$. Each new constraint is checked for consistency with previously generated constraints. A contradiction is fatal: the clause cannot be ordered for sequential execution.

Continuing with the previous example, `L2` induces $G1 < G3$ and `R1` induces $G3 < G2$. So far, all these constraints are consistent. However, analyzing the second set of modes, we find that `X` induces $G2 < G1$. This alone causes the analysis to fail (and subsequent examination of `L1` causes additional contradictions within the second set of modes itself).

If we can derive that the mode of the head of the first argument is 'in' (for example using the context information, as shown in Section 4.2), then there is only one set of modes, and the ordering algorithm terminates successfully with the following order:

```
q([X|L],R0,R) :- s(L,X,L1,L2), q(L2,R1,R), q(L1,R0,[X|R1]).
q([?|?], ^,?) :- s(?,?, ^, ^), q( ?, ^,?), q( ?, ^,[?| ?]).
```

6 Implementation Issues

In this section we will show how the results of the mode analysis can be helpful to improve the implementation of committed-choice languages. In general, straightforward implementation of committed-choice languages has several inefficiencies, such as a large amount of processes, and the requirement for a high memory bandwidth. Furthermore most committed-choice languages are implemented using an abstract machine based on the original WAM [22]. Although this is without any doubt a very good starting point, it becomes obvious that there are several deficiencies in this model when it is modified and applied to committed-choice languages. Our

[11]The first clause has only one goal, so it does not require ordering.

ultimate goal is therefore to define a general abstract machine model which is more at a RISC-level, which, combined with the optimization techniques discussed in this section, will give a speedup compared to previous implementations.

It can already be seen that one of the advantages of the mode analysis is the detection of sequential modules. These sequential modules can be implemented as if they were part of a standard procedural language, using all available and well-understood compile-time optimization techniques. Issues to look at are, for example, better register usage, lower memory usage, and thus the production of less memory garbage.

Currently our aims for efficient compilation are the use of continuation-based goal management, intra-module register allocation (as opposed to intra-procedural register allocation), conversion of unification to assignment (and, if possible destructive assignment), and local memory reuse.

6.1 Continuation-based Goal Management

One of the advantages of the detecting of sequential modules is that these modules can be implemented more efficiently than general "parallel" modules. One of the areas of improvement is the management of goal records. The simplest and most efficient technique of goal management is continuation stacking [1]. This technique is used in the implementation of nearly all conventional languages, such as C and Pascal, as well in most WAM-based Prolog implementations [22]. In contrast, all committed-choice implementations we know of (such as JAM [4], Strand [7], FCP [9], and FGHC [10]) use separate goal records for each body goal. The advantages of using continuation stacking are that no separate full goal records are needed for each and every body goal, and that the goals can be managed as a stack instead of as a heap.

The potential of suspension still exists, so a true stack cannot be built (it could be built on the heap, but since it might suspend, we cannot guarantee that when resumed the top of the stack will not be buried within the heap). Therefore we say continuation "stack," meaning a conceptual stack built of linked records. Such a stack implementation loses the advantage of quick allocation/deallocation, but retains the advantage of minimizing the number of allocated frames. As we established in Section 2, there can be no deadlock due to suspension, as all suspensions of sequential modules are external.

Compare this to committed-choice implementations to date, where "goal stacking" is used to the exclusion of all other allocation methods. This requires the creation of a new goal record for every body goal (except for the last one, if tail-recursion optimization is used) which must be enqueued separately. If the continuation frames are allocated from the heap and are managed as a conceptual stack, then suspension/resumption is straightforward. This scheme supports an unlimited number of suspended "stacks," i.e., sequential modules. These represent large granules of computation beyond which the programmer does not wish to exploit parallelism, because of, for example, a limited number of processors.

6.2 Register Allocation

Normally, almost no register allocation methods can be applied to WAM-based implementations of logic programming languages, because the arguments for a procedure of arity n are forced to be in registers 1 to n. Therefore traditional and more advanced register allocation methods (for example, coloring [2] and interprocedural register allocation [14]) are not applicable here.

The advantage of sequential modules is now that it enlarges the scope within which a register allocator can work from nothing to the entire module. The only restriction on the use of registers is that arguments of the module entry-point goal are still in the first n registers. In this way, it is still possible to retain incremental compilation (although this form of compilation will probably have to be abandoned anyway, in favor of some form of flow analysis). One can even envision an extension of Wall's algorithm [21], where the register allocation is performed at link time, to make it possible to compile logic programs as separate modules (not necessarily sequential), and perform the register allocation when the different parts of a program are loaded.

Because it is guaranteed that invocation of a sequential module only occurs at the module entry-point, and that no calls are made to predicates which are outside the module, a register allocation algorithm does not need to make any assumptions about the placement of arguments in registers other than about the arguments which are received through the module entry-point.

As a simple example of the use of register allocation (even at the WAM level) consider the following module, which is always called through `sumlist/2` with the first argument bound to a list and the second argument unbound, as shown by the mode analysis: `m(<sumlist,1>) =` in and `m(<sumlist,2>) =` out.

```
sumlist(X,Sum) :- true | sum(X,0,Sum).

sum([],Old,Sum) :- true | Sum = Old.
sum([X|Xs],Old,Sum) :- true | New is Old + X, sum(Xs,New,Sum).
```

Using the ICOT PDSS/KL1 compiler system, we get the following KL1 code [10].

```
predicate  sumlist,2
L1:     try_me_else    L2
        put_value      3,2
        put_integer    2,0
        execute        sum,3
L2:     suspend        sumlist,2

predicate  sum,3
S1:     try_me_else    S2
        wait_nil       1
        get_value      3,2
        proceed
S2:     try_me_else    S3
        wait_list      1
        read_variable  4
        read_variable  1
        add            2,4,2
        put_value      1,5
        execute        sum,3
S3:     suspend        sum,3
```

Now using the information from the mode analysis that this module is self-contained (i.e., no calls to predicates outside the modules and all calls to the modules enter at the top-level `sumlist/2`), an optimizing compiler can make the following (simple) optimization:

```
predicate  sumlist,2
L1:     try_me_else    L2
        put_integer    3,0
        execute        sum,3
L2:     suspend        sumlist,2

predicate  sum,3
S1:     try_me_else    S2
        wait_nil       1
        get_value      3,2
        proceed
S2:     try_me_else    S3
```

```
        wait_list      1
        read_variable  4
        read_variable  1
        add            3,4,3
        execute        sum,3
S3:     suspend        sum,3
```

The optimized version uses the knowledge that this module is self-contained by rearranging the arguments of sum/3 such that they don't occupy registers 1 through 3 any more. In the optimized version, the number of instructions executed for an input list of size n decreases from $8n + 8$ to $7n + 7$ instructions. Although not much, a simple optimization already reduces the number of instructions executed by 12%. For larger modules, the gain will be even higher.

Thus intra-module allocation, a limited form of global allocation, can conceivably reap great reduction in memory traffic and signification improvement in performance.

6.3 Memory reuse

A third area for application of the mode analysis and its ability to detect single-producer single-consumer streams is the area of memory reuse. Reuse of memory is important as committed-choice languages consume memory at a high rate, and produce much inaccessible cells, which can only be reclaimed by a garbage collector. An example of this is, for example, the handling of large data structures, where for every update, a new, slightly modified copy has to be made and the old structure is discarded. Therefore it is important to avoid copying structures as much as possible and for an efficient language implementation, it is essential to find techniques to update data structures destructively, i.e., in place, which techniques are as efficient as in procedural languages (or, e.g., as rplaca in Lisp).

The important thing in destructively updating a data structure is knowing when it is safe to do so, i.e. to know when the old version is not required any more. This can be difficult in a parallel environment, but in our sequential modules, it is much easier.

Another, simpler approach for memory reuse is the (partial) reuse of simple data structures, already allocated on the heap, such as a list cell, or a list cell and its head. This approach, as described by Foster [6], uses a specialized 'Reuse'-register and adds some new instructions to a basic WAM-like abstract machine for Strand. However, it assumes the capability of detecting single-consumer streams and is only applicable for reusing data structures within individual process definitions. By using our mode analysis technique and sequential modules, it is possible to extend Foster's technique to cover an entire module.

7 Conclusions

This paper presents a compile-time technique for determining if a set of procedures within a parallel program can be executed sequentially without causing deadlock. More specifically, the analysis and methods are described in the context of committed-choice parallel logic programming languages, such as FCP, FGHC, and Parlog. These concurrent languages have inherent fine-grain parallelism, so that the task at hand is to throttle high-overhead parallelism, rather than uncovering more parallelism. We present a framework of sequential program modules and basic blocks that can be derived and guaranteed to be deadlock-free at compile time. This paper outlines a source-to-source code optimization that ensures that sequential execution can proceed smoothly. Thus "traditional" procedural language optimizations that have previously been discarded by those implementing committed-choice languages (e.g., [4, 10, 16, 17]) can now be considered, such as continuation-based goal management and interprocedural register allocation.

Future research in this area includes two major targets. First, algorithms must be designed for the selection of sequential modules and basic blocks. Selection walks a fine line between making modules too large, thereby throttling too much parallelism, and making modules too small, thereby not increasing granularity enough to be effective. The second target of future research is to gain experience with the techniques by automating the mode-analysis algorithm, and characterizing some benchmark programs. Automation of Ueda's method is an open research area, as is the characterization of what percentage of code within real programs can be successfully sequentialized.

Our ultimate goal is the development of higher-performance parallel logic programming systems, in particular committed-choice languages [16], and languages such as Pandora and Andorra, using an efficient, state-of-the-art compiler system combined with a RISC-like abstract machine, where the addition and deletion of instructions is solely based on their merit for the overall performance. Other parts of our work have, e.g., led to the development of an algorithm for the determinacy analysis of Flat Pandora programs [11], and several parts are currently being implemented.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading MA, 1985.

[2] F. Chow and J. Hennessy. Register Allocation by Priority-based Coloring. In *SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232, June 1984.

[3] K. L. Clark and S. Gregory. A Relational Language for Parallel Programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 171–178. ACM, Portsmouth NH, October 1981.

[4] J. A. Crammond. *Implementation of Committed-Choice Logic Languages on Shared-Memory Multiprocessors.* PhD thesis, Heriot-Watt University, Endinburgh, May 1988.

[5] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 174–188, June 1990.

[6] I. Foster. Copy Avoidance through Local Reuse. Technical Report MCS-P99-0989, Argonne National Laboratory, August 1989.

[7] I. Foster and S. Taylor. Strand: A Practical Parallel Programming Language. In *North American Conference on Logic Programming*, pages 497–512. Cleveland, MIT Press, October 1989.

[8] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation.* Addison-Wesley Ltd., Wokingham, England, 1987.

[9] A. Houri and E. Shapiro. A Sequential Abstract Machine for Flat Concurrent Prolog. Technical Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot, Israel, July 1986.

[10] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, August 1987.

[11] M. Korsloot and E. Tick. A Determinacy Testing Algorithm for Nondeterminate Flat Concurrent Logic Programming Languages. In *International Conference on Logic Programming.* Paris, MIT Press, June 1991.

[12] M. Korsloot and E. Tick. Detection of Sequential Modules in Parallel Programs. Technical Report CIS-TR-91-09, University of Oregon, March 1991.

[13] C. S. Mellish. Abstract Interpretation of PROLOG Programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horwood Ltd, Chichester, 1987.

[14] J.M. Mulder. Inter: An Inexpensive Inter-procedural Register Allocator. In *Proceedings of the 15th Symposium on Microprocessing and Microprogramming*, September 1989.

[15] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[16] E.Y. Shapiro, editor. *Concurrent Prolog: Collected Papers*, volume 1,2. MIT Press, Cambridge MA, 1987.

[17] S. Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[18] E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. *New Generation Computing*, 7(2):325–337, January 1990.

[19] K. Ueda. personal communication, November 1990.

[20] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *International Conference on Logic Programming*, pages 3–17. Jerusalem, MIT Press, June 1990.

[21] D. Wall. Global Register Allocation at Link Time. In *SIGPLAN Symposium on Compiler Construction*, pages 264–275, June 1986.

[22] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.