Experience with a Clustered Parallel Reduction Machine

M. Beemster; P. H. Hartel; L. O. Hertzberger;
R. F. H. Hofman; K. G. Langendoen; L. L. Li;
R. Milikowski; W. G. Vree;
H. P. Barendregt; J. C. Mulder;

January 11, 1993

Abstract

A clustered architecture has been designed to exploit divide and conquer parallelism in functional programs. The programming methodology developed for the machine is based on explicit annotations and program transformations. It has been successfully applied to a number of algorithms resulting in a benchmark of small and medium size parallel functional programs. Sophisticated compilation techniques are used such as strictness analysis on non-flat domains and RISC and VLIW code generation. Parallel jobs are distributed by an efficient hierarchical scheduler. A special processor for graph reduction has been designed as a basic building block for the machine. A prototype of a single cluster machine has been constructed with stock hardware.

This paper describes the experience with the project and its current state.

1 Introduction

Functional programming is founded on the lambda calculus, which is a mathematical theory that provides a sound basis for work on reduction machines [5]. This is particularly important for work on parallel systems, where correctness and reliability are even more difficult to achieve than on sequential systems. The availability of a sound theoretical basis is a significant advantage of functional programming over imperative programming. It allows the implementation to perform a large variety of program transformations aimed at a good mapping of the application onto the available hardware. Compilers for imperative languages also use program transformation for optimisation purposes, but since such languages are not referentially transparent, there is less scope for wide ranging transformations. Purely functional languages are referentially transparent, which means that any well formed expression from a functional program has a well defined meaning that cannot be altered by evaluating the expression [16]. Any reference to the expression will thus always yield the same value, hence the term referential transparency. Because of the use of assignments, this is generally not the case in imperative programs, where the meaning of an expression can often be altered by changing the state of the system.

The disadvantage of functional programming is that the speed of available systems is lower than that of their imperative counterparts. This is not surprising because the development of compilation techniques and hardware for imperative languages have a longer history than
functional languages. However, there is a continuing trend of improvement in the quality of compilers for functional languages and there are indications that implementations of functional languages will catch up with those of imperative languages [30, 52, 51, 37].

In a previous project [6] a two pronged attack was launched on this disadvantage. The first line of research developed a practical computational model (term graph rewriting [62]) as a basis for a high performance compiler of the functional language CLEAN [9, 49]. The second line of research developed a coarse grain parallel evaluation method for functional programs [20, 66] and a prototype architecture.

In the current research programme of the Universities of Amsterdam and Nijmegen, further work has been done to produce faster implementations of lazy functional languages. This paper surveys the results we have obtained so far. A survey of recent work done by other research groups may be found in [36].

Algorithms

1. Methodology
   - divide and conquer algorithms
   - sequential input and output

Lazy programs

2. Compilers
   - strictness analysis
   - dependency analysis
   - insert typing information

Strict programs

3. Code Generators
   - RISC code
   - VLIW code
   - G-hinge code

Machine code

4. Architecture

5. Processor RTS

5. Cluster RTS

5. Machine RTS

Figure 1: Project structure chart

A hierarchical decomposition of the work on the implementation of a parallel functional programming language is shown in the schema of Figure 1. Independent research issues are singled out as separate components, such as work on parallel algorithms, compilers and
code generators. The schema focuses on the major problems, without losing sight of the relationships:

1. **Programming methodology for developing parallel functional programs.**
   Work is in progress on a set of guidelines that can be followed to write good applications for a specific class of parallel reduction machines: scalable machines with a distributed memory. Scalability is the most effective method to increase computing power, as processing and memory units can be added at will.

   At the application level, parallelism is based on the divide and conquer paradigm. Many divide and conquer algorithms have been implemented as part of a parallel functional benchmark. Significant effort has been put into the development of transformational methods to enable synchronous process networks to be implemented as divide and conquer applications.

   A method for performing input/output and process control has been designed for a sequential system that maintains all advantages of functional languages and yet allows the definition of I/O behaviour in a clear and concise style. The method is a refinement of the Haskell [29] approach using predefined functions on opaque objects.

2. **Compilation techniques for functional languages.**
   The developments in high performance compilation techniques for functional languages have come a long way since Turners seminal work [60]. For example, strictness analysis (see section 5.2) has made major advances possible. Existing techniques for imperative languages are also used and even extended beyond what is possible for imperative languages because of the referential transparency of functional languages. The FAST compiler [21] has been designed to study a framework for integrating high level program analysis and synthesis techniques.
3. Code generation techniques for RISC and VLIW architectures.

Two aspects of code generator design are described in some detail here. The first is code generation for RISC processors. The FCG code generator [37] is a back end for the FAST compiler, that performs low level optimisations, such as tail call elimination specifically for RISC architectures.

The second aspect concerns code generation for very long instruction word (VLIW) processors. The Stoffel compiler/code generator [7] has been developed to study low level code optimisation techniques such as register allocation and instruction scheduling, for VLIW architectures. The Stoffel compiler is not based on FAST. It was found that the ability to generate good VLIW-code is highly dependent on the form of all intermediate levels of the compiler. Unfortunately time did not permit to merge these concerns for code generation into the FAST compiler.

4. Systems architecture.

A parallel architecture has been developed as a testbed for the developments at various levels of the system. The architecture has three levels of parallelism. The top two levels exploit coarse and medium grain parallelism, which are both visible to the programmer. This part of the system will be referred to as the macro parallel machine. The bottom level exploits fine grained parallelism, which is not visible to the programmer. This part of the architecture will be referred to as the micro parallel machine.

The structure of the macro parallel machine is shown in Figure 2. The machine consists of a number of clusters that are connected by a high speed network. Within each cluster a number of processors are connected to a shared memory. The ensemble of clusters thus constitutes a scalable distributed memory machine, while each cluster can be viewed as a shared memory machine. This two tier system has implications for the way applications are programmed for the machine. Within a cluster medium grain parallelism is adequate but in the machine as a whole only large grain parallelism is acceptable. The programming methodology takes the differing grain sizes into account.

Different types of processors can be used as processing elements in the clusters: RISC processors (Motorola 88000) and VLIW processors. Based on VLIW principles, a special graph reduction processor has been designed: the G-hinge [45]. The VLIW processors introduce extra opportunities for exploiting parallelism that is not visible to the programmer. This third form of parallelism (micro parallelism) is discovered and used by the code generator.

5. Efficient runtime support.

For true scalability the scheduling of jobs in a distributed memory machine must be controlled in a distributed fashion. The problems associated with distributed control are solved by a hierarchical scheduling strategy.

In the next section the choice for a scalable architecture is motivated. In section 3 a class of algorithms is identified that can be implemented successfully with coarse grain parallelism. The programming methodology based on annotations and transformations describes how these algorithms can be implemented. The second component of the programming methodology is described in section 4, where we describe how input/output facilities can be added to a purely functional language without sacrificing the referential transparency that is needed for the program transformations. Compilation and code generation techniques for the individual processing elements of the parallel system are the subject of sections 5 and 6. Sections 7 and 8 discuss some of the details of the parallel architecture. The last section discusses some of the remaining problems and present the conclusions.
Scalability, both in hardware and software, is an important issue in the design of high performance systems. Scalability in hardware is generally provided by architectures with a distributed memory, which is interconnected by a communication network. Only coarse grain parallel applications with little inter processor communication can execute efficiently on such architectures.

Functional languages provide abundant implicit parallelism, but the fine-grain nature does not match with the scalability requirements. Although efforts have been undertaken to automatically increase the size of basic computation grains [28] no satisfactory results have been presented. Therefore we have adopted the solution of program annotations, to indicate which expressions can be evaluated in parallel. The programmer has to explicitly insert these annotations in the program source, and is responsible for controlling the size of the parallel jobs. A job is thus an expression that has been annotated so that at runtime it may be evaluated in parallel to other jobs.

There are many other ways of generating parallelism from lazy functional programs. Implicit, compiler derived parallelism is used by the AMPGR machine [19] and the HDG machine [34]. Parallelism annotated by the programmer is used in the $\mathit{H/G}$ machine [3], the GAML machine [44], the PAM machine [43], the PABC machine [50] and the GRIP machine [55, 54]. A survey of these recent designs may be found in [36]. Early parallel graph reduction machines have been described in [59, 32, 64].

2.1 Conditions for successful job based reduction

Any expression can be annotated, but parallel evaluation is only beneficial if the jobs satisfy certain constraints (the so called job conditions):

(a) A job has to be self contained, that is the runtime representation of the job must not contain references to other jobs. This allows a job to be evaluated in a separate address space and avoids the need for garbage collection across jobs. In section 7 this constraint is relaxed to allow expressions that are evaluated in the same cluster to share common subexpressions.

(b) The final result of the program cannot be computed unless the the job is fully evaluated. This condition makes sure that the result of a job is essentially used in the computation as a whole and so no processing power is wasted. Speculative parallelism is thus not considered.

(c) The cost to evaluate a job must outweigh the cost incurred in allocating the job to an available processor. This requirement guarantees that parallel execution of a set of jobs will be faster than their sequential execution.

A programming paradigm that fits these conditions is the divide and conquer paradigm. It partitions a problem into independent parts that are ideal candidates for the job annotation. A large class of applications can be executed efficiently with straightforward divide and conquer parallelism, while transformational methods have been developed to cover synchronous process networks and pipeline parallelism as well [67, 39]. The lazy evaluation mechanism, however, complicates parallel execution of jobs since the “independent” coarse-grain expressions as annotated by the programmer may refer to shared, delayed computations at the graph reduction implementation level. In sequential implementations such delayed computations are simply updated with the result, so all references to the suspension can share the result of the (single) evaluation. In parallel systems special measures have to be taken to support the
sharing of results. The sandwich parallel reduction strategy [66] has been developed to aid in writing programs that meet job condition (a). It is the programmers responsibility to guarantee that the remaining job conditions are also met.

2.2 The sandwich reduction strategy

The sandwich reduction strategy handles subexpressions that are shared between jobs. Instead of copying common shared subexpressions or guarding them with locks for exclusive access, the sandwich strategy reduces the shared computations to normal form before starting the jobs. During compilation, an annotated program is transformed (see section 3) into an equivalent program where the annotations have been replaced by explicit calls to the sandwich primitive, which implements the sandwich strategy. The sandwich primitive has the following form:

\[ \text{sandwich}\ G\ job_1 \ldots job_n \]

where

\[ job_i = F_i\ a_{i1} \ldots a_{im_i} \]

and

\[ F_i\ \text{and } G\ \text{are arbitrary functions} \]

An arbitrary expression is sequentially reduced to normal form until an application of the sandwich primitive is encountered. Then the following steps are taken:

1. All shared expressions are “squeezed” out of the jobs. This means that the function bodies \( F_i \) and their corresponding arguments \( a_{i1} \ldots a_{im_i} \) are each evaluated to full normal form (i.e. not just head normal form). Reducible expressions within function bodies are also fully normalised.

2. A set of jobs is sparked to evaluate the arguments of \( G: job_1 \ldots job_n \) to full normal form and in parallel.

3. Upon termination of all jobs from step 2, the function \( G \) is invoked with the computed argument values. Then normal order reduction resumes.

The squeeze in step 1 guarantees that the data shared between jobs is always evaluated so, for distributed systems, jobs can be copied safely to remote processors without duplicating work, while for shared memory systems data can always be accessed without locking for exclusive access. The disadvantage of the squeeze is the deviation of the standard lazy evaluation mechanism, which might lead to non-termination in rare cases. In [39] a set of rules is given for the programmer to transform such programs into terminating equivalents.

3 A methodology for parallel functional programming with annotations

There are many types of parallelism (see [25] for a classification). The sandwich reduction strategy supports only those types whose communication structure is synchronous: a task that has executed a sandwich primitive is suspended until all its children have terminated, thus a task cannot execute concurrently with its children. The sandwich has fork-join semantics, as opposed to spark-and-wait. Divide and conquer applications are eminently suitable for the sandwich strategy.

Due to the referential transparency of functional programs, semantics preserving program transformations are not difficult to apply. A methodology has been developed to transform
another class of parallel applications to fit the sandwich semantics as well. These applications are synchronous process networks. A process network is a set of recursive equations over lists, where function applications are viewed as processes and the lists represent the connections between the functions [31]. Geometric parallelism or data parallelism belongs to the class of process networks. Process networks may be cyclic, in which case previous elements of a list are necessary to compute further elements of the same list. A process network is synchronous if for each process in the network, static analysis is sufficient to determine the number of input elements required to produce the next output element. Pipe-line parallelism, for instance, is supported only insofar as the stages in the pipe-line behave in a lock-step fashion, each stage producing a predictable number of outputs and consuming a predictable number of inputs. For applications where the production or consumption of stages within the pipe-line is not compile-time predictable (this is the case in the standard parallel implementation of Eratosthenes’ sieve), the sandwich primitive for parallelism cannot be used. The transform methodology for synchronous process networks results in a run-time job behaviour where parallel phases alternate with global synchronisation phases. With Amdahl’s Law [24] in mind, these global phases should take a short time in comparison with the parallel phases: only coarse-grained process networks are suitable.

Given a suitable parallel application, possible parallel jobs are annotated as such by the programmer and a set of program transformations are applied to generate an efficient implementation, with further help by the programmer to make decisions about the grain size of parallel jobs. A number of additional transformations have been developed specifically to support coarse grain parallelism. These are applied together with some standard transformations described in the literature (e.g. [8, 11, 15]). The following sections show the major aspects of program development by discussing some examples of parallel functional programs. The first three examples are divide and conquer algorithms, that differ in the way the grain size is made suitable for the architecture. The fourth example is a process network.

3.1 Quicksort

Quicksort is the standard example of a divide and conquer algorithm. The program is shown here as a Miranda\(^1\) literate script [61]. Subexpressions that can be evaluated in parallel are annotated by the programmer using angular brackets (\(\{\) and \(\}\)). Angular brackets obey the same syntactic rules as the normal parentheses, except that an expression between matching angular brackets is a job. Angular brackets are not part of Miranda. The efficiency of the program has to some extent been sacrificed to avoid clutter in the presentation.

\[
\begin{align*}
> \text{qsort0} &\ [\ ] = [\ ] \\
> \text{qsort0} &\ (a:as) = \{ \text{qsort0} \, ls \} ++ (a: \{ \text{qsort0} \, rs \}) \\
> \text{where} \\
> &\ (ls,rs) = \text{qsplit} \, a \, as \\
> \text{qsplit} &\ p \, as = ([a \mid a<-as; a < p],[a \mid a<-as; a >= p])
\end{align*}
\]

A program annotated with job brackets can be transformed more or less automatically into a version with sandwich expressions. A formal description of the transformation may be found in [66]. Here the ideas of the transformation will be shown by means of a series of examples. The transformation requires two steps. The first step, which is called job lifting, recognises expressions between job brackets. Job lifting generates an auxiliary function combine, to avoid the application (qsort0 rs) from being evaluated too early. In the definition of qsort1 (see below), job lifting has replaced the body of qsort0 by an application of combine:

---

\(^1\)Miranda is a trademark of Research software Ltd.
For the sandwich strategy to be effective, both recursive applications of \texttt{qsort1} should contain enough work to outweigh their communication cost (cf. job condition \(c\)). This may be achieved by imposing a lower limit on the length of the lists \(ls\) and \(rs\). The next version \texttt{qsort2} (below) shows how controlled application of the sandwich strategy can be obtained by a second transformation step, which is called the grain size transformation. When the grain size drops below a certain threshold \(t\), the program switches to the original sequential version \texttt{qsort0}. The length \((n)\) of the list to be sorted is taken as a measure of the grain size, since the amount of work is \(O(n \times ^2 \log n)\). In the final version (not shown) some redundant calculations are removed by standard transformations. Although the final version \texttt{qsort2} has a complex appearance, it should be noted that most of the code is generated by two program transformation steps. Most of the work contained therein can be automated, but not without guidance by the programmer.

The cost involved in the control mechanism that is introduced by the grain size transformation has to be weighed against the benefits from parallel evaluation. The optimal value of the threshold \(t\) depends on properties of the system configuration. This problem is studied in \[66\].

### 3.2 The fast Fourier transform

Unlike the quicksort algorithm the divide and conquer version of the fast Fourier transform perfectly divides the input data into two equal parts at each recursive invocation. This should allow for an optimal processor utilisation. The essential part of the program with the job annotation is shown below, where \texttt{inputsize} is the number of points in the transform (\texttt{inputsize} must be a power of 2) and \texttt{bfly} is the classic butterfly calculation \[14\] with complex numbers:

\[
\texttt{b/}\, y \, n \, x \, y = (x + z, x - z) \text{ where } z = w^n \times y \text{ and } w = e^{2\pi i / \text{inputsize}}
\]

The result list produced by the Fourier transform as shown below is not in the right order and has to be passed through a reorder function, which is not shown here. A comprehensive treatment of the fast Fourier transform may be found in \[23\].
Since \texttt{fft} already requires the length of the list of data as a parameter this information is readily available for the purpose of controlling the grain size. The transformation to the final sandwich version with threshold control is therefore straightforward [66].

### 3.3 Wang’s algorithm for solving a sparse system of linear equations

Many mathematical models of physical reality consist of a set of partial differential equations. An important step in approximating the solution of such a set of equations is to solve a large set of linear equations. The corresponding matrices often appear to be in a tri-diagonal or block tri-diagonal form. Wang has proposed a partitioning algorithm to achieve parallelism in the elimination process of a tri-diagonal system [71]. The basic idea of the algorithm is to divide a tri-diagonal matrix in equally sized blocks and to try elimination of these blocks in parallel. The function \texttt{wang} (below) shows the annotated schema of Wang’s algorithm, which has three phases: the first elimination, the sequential part and the second elimination. The first elimination in a block gives rise to “fill in” outside that block. This fill in has to be exchanged by neighbouring blocks by the sequential part of the algorithm before the second elimination can be done:

```haskell
> wang matrix0 mark = parmap second_elimination matrix2
> where
>     matrix2 = sequential part matrix1
>     matrix1 = parmap first_elimination matrix0
>
> parmap f [a] = [f a]
> parmap f (a:as) = { f a } : { parmap f as }
```

Parallelism is introduced by the function \texttt{parmap}, which takes a list of blocks as its second argument. The grain-size of the parallel computations of this program is completely determined by the size of the blocks into which the matrix is initially divided. In contrast to the previous examples, there is no dynamic grain size control.

The quicksort and Fourier transform require extra code to control the grain size. This causes performance loss, which must be made up by parallelism. The Fourier transform requires less extra code and thus suffers less from the transformation loss than quicksort. Wang’s algorithm does not introduce extra code to control the grain size but requires extra code to lump computations into larger grains. Although all three problems belong to the class of divide and conquer algorithms the implementations have quite different characteristics when it comes to exploiting the parallelism.

How successful the exploitation of parallelism is depends on many factors, such as the number of processors in the machine, the application, its input data set and many more. When properly qualified, a good measure of how successful the exploitation of parallelism has been, is the speedup with respect to the evaluation of a sequential version of the application under consideration, with the same input data set (which is thus not the same as the parallel version running on 1 processor). However, given enough processors, it is easy to achieve a good speedup, e.g. 10× on a 1000 processor machine. This is misleading, and therefore undesirable practice. Instead \textit{economic} speedup figures are used, defined thus: when at least 50% of the total processing capacity has been used, the measured speedup is an economic speedup. There is no point in building a system with many processors that are idle for most of the time.

Compared to the execution of the sequential version of each of the algorithms, economic speedups were found of 2.2 on a 4 processor system for quicksort, 4.5 on 8 processors for the Fourier transform and 2.7 on 5 processors for Wang’s algorithm. The characteristics of the applications, input data sets and other relevant parameters are described in [66].
3.4 Transformation of cyclic process networks

Job lifting and grain size transformation are necessary to enable divide and conquer algorithms to be implemented efficiently on a coarse grain parallel reduction machine. To enlarge the class of algorithms that can be implemented successfully on such a machine, another set of transformations has been developed to turn algorithms based on process networks into divide and conquer programs. The basis of the transformation from a cyclic process network to a divide and conquer algorithm is communication lifting [68]. Consider as an example the set/reset flipflop as shown in Figure 3. The arrows represent streams of bits, which can be implemented as infinite lists of 0s and 1s, as in the Miranda program below:

```
> setreset0 cs ds = as
>   where
>   as = nand 0 ds bs
>   bs = nand 1 as cs
>
> nand s xs ys = maps_2 notand s xs ys
> notand s x y = bnot (band x y)
>
> maps_2 f s xs ys = s : maps_2 f ( f s (hd xs) (hd ys) ) (tl xs) (tl ys)
```

The function `setreset0` takes the two stream arguments `cs` and `ds` as input, where `cs` represents the set pulses and `ds` represents the reset pulses. The two local definitions (`as` and `bs`) represent cyclic connections in the network of Figure 3. At each step, the two `nand` functions calculate the next output bit from the current inputs. This results in a unit time delay on the inputs. Although the streams `as` and `bs` are connected in a cyclic fashion, the state computations in the `nand` functions are not cyclic. This becomes apparent when `nand` and `maps_2` are each unfolded once in the definition of `as`, and also in the definition of `bs`:

```
> as = nand 0 ds bs = maps_2 notand 0 ds bs = 0 : maps_2 notand ...
> bs = nand 1 as cs = maps_2 notand 1 as cs = 1 : maps_2 notand ...
```

Both `nand` functions are able to produce their first output element without even referring to the inputs. To produce the next output, the `nand` functions must exchange their present states, which is why the streams must be connected in a cyclic fashion. The communication lifting transformation in effect separates the communication aspect from the computation of the next states. The communication lifting transformation has been formally specified and also implemented as an automatic program transformation tool [68]. The end result of the transformation is:

![Figure 3: Cyclic streams in the set/reset flipflop](image-url)
The computation now starts with an initial state triple \( s_0 \) and the two input lists \( cs \) and \( ds \). Together with the first input elements \( c (=hd cs) \) and \( d (=hd ds) \) the initial state is presented (by \( maps_2 \)) to the \( nextstate \) function. The two results of the actual “flip-flop” computations are then assembled into a new state \( (a, a', b') \). The result of the application of \( maps_2 \) is thus a list of state triples, such that the next triple is calculated from the previous one and the current two input elements from \( cs \) and \( ds \). The list of triples is transformed into a list of single output values by mapping \( sel_3 \) over the triple list. The tail of the list of output values has to be taken because the computation is started with a dummy output value. As shown above, the computations in the \( nextstate \) function can be annotated with job brackets. This particular example only has fine grain computations that are not likely to make parallel evaluation beneficial.

The communication lifting method has been used for a functional program that implements a mathematical model of the tides in the North Sea [65] and a digital hardware simulator [68]. The transformed version of the tidal model experiences an economical speedup of 2.2 on a 4 processor coarse grain parallel reduction machine of which only 2 processors are used. The speedup exceeds the number of processors used because the transformation improves the sequential program by a factor of 1.2.

Annotations to generate parallelism should always be applied with care and communication lifting is no exception to this rule. In particular when dealing with large networks, one must make sure that most of the tuple elements do require some significant amount of work. Otherwise much time will be spent on constructing the tuples, without any opportunity for parallel work. A good way to deal with a large network is to divide it into a number of smaller networks, and apply communication lifting to each sub-network. After communication lifting, the program can be reassembled and as a whole, it will contain fewer, but larger processes. The whole procedure can be reapplied if necessary to build a hierarchy of communication lifted processes. Communication lifting can thus be viewed as a method for grain size enlargement.

4 A methodology for input/output and process control in a functional context

Pure functional programming constructs are by definition side-effect free. However, input and output are side effects. Therefore, performing input and output seems incompatible with functional programming. On the other hand, a program that does not produce any output is completely useless and a program whose output does not depend on its input has very limited usefulness. In the compromise used in the CLEAN language [9, 49] input and output streams are represented within the program as opaque “objects”. These objects can be manipulated by invoking special predefined functions that take the object as their last argument and return a tuple with a new “version” of the object as the last component. When that has happened, the
old version of the object is no longer valid. This implies that CLEAN is not entirely referentially transparent: evaluating one expression can have the side effect of invalidating a syntactically unrelated expression. Fortunately, most of the useful consequences of referential transparency still hold.

5 Compilation techniques for lazy functional programs

The structure of a front end compiler for a general purpose lazy functional language consists of a number of separate and relatively independent components [52], such as lexing and parsing with error recovery, polymorphic type checking and resolution of operator overloading, general simplification of the program, which includes translation of list comprehensions into ordinary function calls, translation of pattern matching into cases or conditionals, lifting of nested function definitions to global level (lambda lifting), inlining and specialisation based on heuristics.

The core of every functional language (the lambda calculus) is a simple, but powerful language by itself, which contains the essence of all the problems associated with efficient compilation of functional languages. This makes it possible to perform research on parts of the compiler while relying on work by other researchers for the remaining parts, in particular the translation of powerful general purpose programming language constructs into the lambda calculus.

Two topics will be discussed here. The first is the automatic translation of untyped functional program into typed programs. This facility was needed because a substantial investment had been made into a benchmark of untyped parallel SASL programs.

The second topic is the design of a flexible framework to integrate various useful program analysis and synthesis techniques for functional programs. The most important of these techniques is strictness analysis. In the next section the typing transformation is discussed briefly, followed by a discussion of the purpose of strictness analysis and its merits.

5.1 Transformation of untyped programs into programs that can be statically type checked

A functional program written in one lazy functional language can be transformed quite easily into an equivalent program in another lazy functional language, because all lazy functional languages are based on normal order reduction of lambda calculus expressions. An important exception to this rule is the transformation that introduces type checking to an untyped program. To translate (untyped) SASL programs into a strongly typed language such as Miranda requires a non trivial program transformation. Such a transformation has been developed: the type checking transformation. It works by wrapping a generic object type around all the dynamically typed objects normally found in a SASL program. In effect, there is only one type of object now in the program and it can be statically type checked. Type checks will be done at run-time when objects (the real ones inside) must be unwrapped to manipulate them, for example in basic arithmetic functions.

But this is only half the work. All the explicit wrapping and unwrapping causes enormous inefficiencies. Fortunately, most of the wrap/unwrap operations are redundant and can be removed by an optimisation transformation. In the optimal case, a well typed SASL program can be transformed into a typed program without any additional wrapping and unwrapping. In practice this cannot be achieved mainly because in SASL lists are the only available data structure. When lists are used to represent tuples, the lists are often inhomogeneous, so that the list elements must remain wrapped. Experimental results show that after the type check transformation and conversion into LML, a benchmark of programs runs on average only at half
5.2 Program analysis in the FAST compiler

The FAST project [21] has developed a compiler for a simple lazy functional language that performs a variety of program analyses to enable efficient code generation. The compiler is based on flow graphs, which can be regarded as dependency graphs. Flow graphs provide a formal framework for expressing program analyses and code generation in an integrated fashion.

Strictness, boxing and update analysis are program analysis techniques that most compilers for a lazy functional language will perform. Strictness analysis allows a call-by-name evaluation strategy to be transformed into the more efficient call-by-value. Boxing analysis identifies the objects which need to be stored in the heap (in boxed form), so that the remaining objects can be allocated more efficiently in registers, or the stack (in unboxed form). Update analysis determines which suspensions, when evaluated, require an update of the heap cell they are stored in.

All major analyses performed by the FAST compiler are based on a linear non-flat domain [10, 69]. This enables the compiler to reason about strictness, boxing and other properties of structures presented as arguments to functions. The chosen domain allows properties to be inferred about top level constructors and about the structure of lists. This will be illustrated by means of an example. Consider the Miranda function append:

```
> append [] ys = ys
> append (x:xs) ys = x : append xs ys
```

The following strictness properties can be inferred by the compiler about the arguments of the function append:

- **Append** : \(\bot\)\(\rightleftharpoons\) head-strict in first \(\bot\)
- **Append** : \(\bot\)\(\rightleftharpoons\) not head-strict in second \(\bot\)
- **Append** : \([x_1, \ldots, x_n]\)\(\rightleftharpoons\) spine strict in both \(\bot\)
- **Append** : \([\bot, \ldots, \bot]\)\(\rightleftharpoons\) spine-of-head strict in both \(\bot\)

The symbol \(\bot\) can be read as “completely undefined”. Property (1) thus means that if the first argument to append is completely undefined, so is the result, regardless of the value of the second argument. The compiler uses this information to ensure that the first argument is evaluated before append is actually called. This saves time and space because no suspension needs to be built and subsequently evaluated for this argument. Property (2) states that it is not safe to also evaluate the second argument before calling append. The reason is, that it is correct to use append when for instance the first argument is non-empty and the second is undefined thus:

\[\text{hd (append \[1\] \(\bot\)) = 1}\]

Property (3) states that in a context where a finite list is required, both arguments to append can safely be evaluated far enough to develop the full spines of the lists. However, none of the elements of the lists may be evaluated yet. Should the computation on one or perhaps both arguments fail to terminate, so would the entire call to append, which establishes the safety of the method. The last property shown here (4) states that it is safe to evaluate all elements of both lists to head normal form in a context that requires a full list of head normal forms.

Property (3) above is actually a statement about the tail field of the cons cells that make up the input lists for append. The linear non-flat domain does not allow similar statements to be
made about the head field of a list constructor. With the present domain it is thus not possible for the FAST compiler to infer that:

\[
\text{append} \left( \bot : xs \right) ys = \bot : \ldots
\]

The input language of the FAST compiler is similar to Miranda. To use the FAST compiler effectively, the Miranda program development system must be used to develop and debug a program. When development is complete, the FAST compiler is used to generate efficient code.

The output language of the FAST compiler is C, which has been chosen because of its portability. An efficient runtime system is available, which allows statistics to be gathered about the runtime performance. In [21] a break down of the benefits of a number of analyses is presented, each performed at increasing levels of sophistication, and analysed for a set of medium-sized functional programs.

6 Code generation techniques for lazy functional programs

The task of the back end compiler for a lazy functional language is to take advantage of all the information that the front end compiler has been able to gather when generating code for a specific target machine. Two research efforts (FCG and Stoffel) will be described in the following sections. A third research effort (the G-Hinge) is related to the reduction machine architecture and presented in section 8.

6.1 The FCG code generator

The Functional C Code Generator (FCG) is a back end for the FAST compiler. FCG produces efficient code that supports two-space copying garbage collection in combination with divide and conquer parallelism [37]. In contrast to other functional language compilers that generate assembly directly [30, 43, 57], FCG uses the C compiler for target code generation, providing high-quality code optimisations and portability. The input language and the output language of FCG are thus both C. The generated code uses tagged data values and an explicit call stack to support garbage collection and parallel reduction (Section 7).

The FCG code generator is organised as a pipeline of three phases. First, the output of the FAST front end is transformed into virtual assembly (KOALA) for an abstract graph reduction machine that consists of a cpu with an unlimited number of registers, a stack, and a heap. Next, the straightforward stack-based KOALA code is optimised into a register-based equivalent form. Finally, the KOALA code is handed as one large function to the GNU C compiler, which is used as a portable assembler. The C compiler performs register allocation, code scheduling and various low level optimisations like common subexpression elimination.

The first FCG compilation scheme to generate KOALA code is rather simple since the input language, as generated by FAST, is a subset of C: no global variables, single assignment of local variables, if-then-else as the only control structure, and no built in operators, but calls to library functions instead. Hence, in essence FCG has to translate function calls only: evaluate the arguments one by one on the call stack and jump to the function entry. The library functions that perform primitive operations like arithmetic are encoded in KOALA, and operate on tagged data values to enable the garbage collector to distinguish pointers from basic data values like integers and characters. To minimise the tagging overhead the tags are (partly) encoded in the unused least significant bits of pointers to heap allocated data. These tags are also used by the lazy evaluation mechanism to distinguish between (head) normal forms and suspend computations (closures).

Feeding the straightforwardly compiled KOALA code into the C compiler results in poor runtime performance since the C compiler cannot “understand” the meaning of the KOALA
stack instructions and compiles every push and pop instruction into loads and stores. Therefore
the FCG code generator includes several optimisation filters to transform the stack code into a
form that is amenable to further optimisations by the C compiler:

- Inside basic blocks the top of the call stack is stored in temporary register variables;
  instead of pushing a value on the stack it is moved into a fresh register (KOALA provides
  an unlimited number of registers), while the corresponding pop instruction is translated
  into a register move. At the end of a basic block, for example when calling a function, the
  registers that hold the top of the stack are flushed to the real KOALA stack.

- The size of the basic blocks is enlarged to increase the effect of the stack caching, by
  inlining the library functions for primitive operations such as +. Inlining of user functions
  is already provided by the FAST front end.

- To extend the C optimisations across function calls, the parameters are not passed on
  the (physical) stack, but in a few global registers instead. Calling a function amounts to
  storing the arguments in a fresh set of registers, saving the local state on the call stack,
  transferring the arguments to the fixed global parameter registers, and jumping to the
  function entry point.

- To avoid saving/restoring “dead” variables, a lifetime analysis is performed for the
  (cached) stack locations inside a call frame.

- Tail calls, which frequently occur since the FAST compiler does not emit loops, are
  transformed into straight jumps to avoid chains of return sequences.

The use of these optimisations is mandatory for good performance: the optimised bench-
mark programs show a speed-up of 1.9 to 7.5 over the straightforwardly compiled stack code.

A comparative study [22] shows that the code generated by FAST/FCG compiler for a
functional benchmark of a dozen medium size Miranda programs is slightly faster than the
code generated by its competitors, which are the LML and Haskell compilers from Chalmers
university [4, 2], the Glasgow Haskell compiler [58] and the Nijmegen Clean compiler [63].

6.2 Code generation for VLIW processors

Fine grain parallelism in VLIW processor architectures offers a compiler for a lazy functional
language many opportunities for code optimisation. Two separate approaches to exploit VLIW
parallelism are underway. The first is the the G-line (see section 8); the second approach is
based on code generation techniques embodied in the Stoffel compiler/code generator, to be
discussed in the next section. Instruction scheduling and register allocation techniques are
applied to an intermediate level in the compiler. At that point it is possible to make use of the
properties of lazy functional languages to allow more freedom in instruction scheduling and
register allocation. Currently target code is compiled for an ideal simulated VLIW machine,
which has 32 registers and an unlimited instruction word width.

Much of the fine grain parallelism, and this holds in particular for the G-hinge, will come
from parallel memory references. Hence, a successful architecture based on these principles
requires a high memory bandwidth, for example implemented by multiple paths to multiple
memory banks. This is expensive in terms of the cost of the machine architecture, but not un-
heard of for machines that require high memory bandwidths, for example in supercomputers.
Our research is aimed at finding possible gains obtained from fine grain parallelism. When we
can identify such gains we will be able to make an assessment of cost versus performance.
Fine grain parallelism in VLIW processors

The referential transparency of pure functional programs allows the code generator much freedom in scheduling instructions, provided the data dependencies between computations are maintained. This is an advantage of functional languages over imperative languages, because the side effects in the latter severely restrict the possibilities the code generator has to optimise the code.

A VLIW processor can execute a number of operations at the same time. In a pipelined processor the operations may overlap but they must start one after the other. In a VLIW processor a number of operations are packed into one (Very Long) instruction, so that all are started at the same time. The parallelism exploited in a VLIW processor is fine grain parallelism, at the instruction level. There is only one program counter in a VLIW processor, which points to the current instruction. Hence, at this level of the machine there is no notion of parallel processes.

Parallelism in a VLIW processor is completely under control of the compiler. The task of the compiler is to take a (normal) sequential thread of operations, analyse all dependencies in the thread and make a conservative estimate when dependencies are unknown. The compiler applies list scheduling (which is basically topological sorting [17]) to the thread. For example, in an expression like \( r = \frac{(a + b) \ast (c + d)}{c + d} \), the sequential code (using two temporaries \( t_1 \) and \( t_2 \)) would look like this:

\[
\begin{align*}
ADD & \ a \ b \ t_1 \\
ADD & \ c \ d \ t_2 \\
MUL & \ t_1 \ t_2 \ r
\end{align*}
\]

Dependency analysis and list scheduling will find that the two additions can be scheduled in the same instruction. The code becomes (where // means “in the same instruction as”):

\[
\begin{align*}
ADD & \ a \ b \ t_1 \ // \ ADD \ c \ d \ t_2 \\
MUL & \ t_1 \ t_2 \ r
\end{align*}
\]

Dependencies are not the only limiting factor that prevent the compiler from moving all operations into one instruction. Another limit is the number of functional units. In the previous example the compiler must also make sure that there are indeed two functional units that are able to do the additions, and also specify which functional unit does what. In a VLIW processor there are no provisions for making these decisions in hardware at run time.

The advantage in having a functional language as the source of a translation are in the dependency analysis, and specifically aliasing analysis. Aliasing analysis is used to find (in)dependencies of memory reference operations. If a read and a write operation refer to the same address in memory, their order is important, hence there is a dependency between the two operations. This restricts scheduling opportunities. Without information about the precise addresses, the compiler has to make worst case assumptions. This is especially wasteful since most memory references do not address the same location. Aliasing analysis tries to find out when two memory references do not refer to the same memory location. In the general case (for example for C programs) this kind of pointer analysis is hard and often intractable [1]. For functional programs, it is known that there are no side-effects and that memory is written only once. This allows the dependency analysis to make stronger assumptions. Updating of evaluated suspensions is a side-effect and must be treated as a special case.

The Stoffel compiler first translates a lazy functional program into an intermediate form similar to the spineless tagless G-machine [53]. It then generates VLIW code. The basic ideas behind the compilation to VLIW code can be shown by means of the following program fragment, which is typical of the translated code in the spineless tagless machine:
PUSH a 
PUSH b 
PUSH c 

A straightforward compiler would translate this for a RISC like processor into:

STORE a (sp,0) -- Store register a at address sp+0 
SUB sp 1 sp -- Decrement stack pointer by 1 
STORE b (sp,0) 
SUB sp 1 sp 
STORE c (sp,0) 
SUB sp 1 sp 

It is a waste of time to decrement the stack pointer 3 times in a row. It is also important to note that this instruction ordering totally constrained by the dependencies between the USEs and DEFINes of the stack pointer in every instruction. An instruction scheduler cannot do anything to introduce parallelism into this sequence. The instruction scheduler is allowed much more freedom if instead the compiler, by using a form of constant propagation, translates the three PUSH instructions into:

SUB sp 3 sp 
STORE a (sp,3) 
STORE b (sp,2) 
STORE c (sp,1) 

In this case the dependency graph contains arcs between the first and each of the other three instructions. Aliasing analysis can easily find out that the addresses sp+3, sp+2, and sp+1 are all different, hence there are no dependencies between the three STORE instructions. If the architecture provides at least three separate access ports to the memory, maximal parallelism can be introduced into this example. The code becomes:

SUB sp 3 sp 
STORE a (sp,3) // STORE b (sp,2) // STORE c (sp,1) 

Although this example is simple, the same method can be applied to graph/closure building and is very important for reducing explicit sequencing in the dependency graph. The same transformation would hold for imperative programs, but there kind of code sequences occur less often and hence are less significant for total performance.

Register allocation

Register allocation is as important to a lazy functional language compiler as it is to a compiler for any other language. Register allocation and instruction scheduling for VLIW depend on each other [13, 18]. To obtain optimal results, register allocation and instruction scheduling should be done at the same time. This is complicated and compute intensive. In the software pipelining technique [35] register allocation is performed after instruction scheduling. The instruction scheduler assumes that enough registers will be available.

In the Stoffel compiler register allocation is performed first, before instruction scheduling. In this way schedules that cannot be satisfied with a limited number of physical registers are not even generated. The compiler assumes that there is an unlimited number of virtual registers. For every new variable or temporary used, a fresh unique virtual register is allocated. This results in a large number of virtual registers with a short lifetime. It is important, that USEs and DEFINes of two different virtual registers are independent of each other. If the dependency
graph would be built at this moment, no dependency arcs between these different virtual
registers would be present. After register allocation different virtual registers may be mapped
onto the same physical register thereby introducing “false” dependency arcs. This reduces
freedom in instruction scheduling. To minimise the harm done by these false dependencies
the Stoffel register allocation scheme uses as many physical registers as possible. Allocation
of registers is in a cyclic/round robin fashion. This causes a freed register not to be used in
the immediate vicinity of its last use. It turns out that this mechanism works very well. The
instruction scheduler has much freedom in packing code sequences into multi operation VLIW
instructions. When the instruction scheduler would not be able to benefit from the potential
resources of the VLIW processor because of lack of registers, this points towards an unbalance
in the hardware which could be solved by adding more registers.

**Status of the VLIW code generator**

Code generated in the Stoffel compiler has a simple basic block structure. This is inherent to
the structure of functional languages, which have no loops, only recursion. There are forks and
joins in the thread of a function. These are introduced by the CASES (as compiled for pattern
matching) and conditionals. Because of this simple block structure, the register allocator can
operate on a whole function at a time.

To obtain an indication of the parallelism in a functional program that can be exploited in
a VLIW architecture we will now look at the code generated for the function `append`, in the
form that is obtained after translation of pattern matching into CASE expressions:

\[
\text{append } xs' \ ys = \text{CASE } xs' \ OF \ \\
\quad <\text{NIL}> = \ ys \ \\
\quad <\text{CONS} \ x \ xs> = x : \text{append } xs ys \ \\
\text{ESAC}
\]

The (virtual) VLIW-machine for which this function is compiled has 32 registers, no limit
on the amount of operations in a single instruction, single cycle instruction execution and no
limit on the memory bandwidth. The function `append` has three basic blocks. The first is the
evaluation of the argument `xs'` to head normal form. The second basic block is executed when
the `<NIL>` case applies, the third when the `<CONS x xs>` case applies. The evaluation of `xs'`
in the first basic block contains 4 operations, which are largely sequential:

```
SPIN r3    -- load first argument xs' into register 3
SPIN r4    -- load second argument ys into register 4
EVAL r3    -- evaluate cell pointed at by register 3 to hnf
LDX r3 1 r5 -- load tag from the result of eval into register 5
```

The second basic block builds a node for `<NIL>`. The block contains 8 operations, which the
scheduler packs into 5 instructions. The first instruction contains 2 ALU operations, the second
1 ALU operation, the third 3 (2 memory + 1 ALU operation) and the last two instructions
contain 1 ALU operation each. This gives an average parallelism of 8/5=1.6 on a system with
at least 2 ALU and 2 memory units.

The third basic block exploits more of the capabilities of VLIW-code. It contains 20 op-
erations that together build a list-node and a closure-node for the delayed recursive call to
`append`. These operations are scheduled into only 6 instructions. This results in an average
parallelism of more than three on a system with at least 6 ALU units and 4 memory units.

In the case of `append`, the amount of parallelism is limited by true data-dependencies
between operations. In larger functions, for example with LETs that build large expression
graphs, the scheduler puts many more operations into a single instruction. Some benchmark
programs allow up to 12 operations to be packed into a single instruction.
Parallelism in graph reduction occurs when the graph has more than one reducible expression. These can each be reduced by a processor, and the intuitive architecture model is therefore a shared memory multiprocessor, where the processors are busy rewriting their private part of the graph. However, shared memory systems are not scalable: bus contention becomes a bottleneck when more processors are added, although caches can stretch the limit. As one of the primary design goals for the macro-parallel machine is scalability, it must have a distributed memory architecture. For increased performance, each node in the distributed memory multiprocessor is itself a shared memory machine. The network that connects clusters will be a state of the art point-to-point network. The macro parallel machine is thus a two level architecture. Each level has its specific resources and corresponding run time support system to manage them.

The term task will be used exclusively for (medium size) parallel grains that cannot be evaluated outside the cluster in which they are created. The term job is reserved for the (coarse) parallel grains that may be evaluated anywhere. Both tasks and jobs are generated by the sandwich strategy, so the programmer is responsible for creating jobs and tasks. The programmer also decides, which parallel grains are jobs and which grains are tasks.

The runtime support system for the shared memory clusters controls synchronisation between tasks, manages storage and schedules the processors in the cluster with help of a global task list. The runtime support system for the scalable distributed memory machine manages the network for graph transport and control messages and distributes jobs over the clusters with a distributed scheduling strategy. The two runtime support systems are completely independent, but the design of both is strongly interwoven with the semantics of the sandwich strategy for parallelism, see section 2.

7.1 Inter cluster run time support system

The distributed memory machine is scalable. This implies that the scheduling algorithm must be distributed, because a central scheduler would become a bottleneck. A hierarchically distributed scheduling algorithm seemed most suitable, because flatly distributed control algorithms have a control integrity problem: such schedulers react independently of each other, so situations occur where many schedulers respond to a local perturbation that should have been resolved locally. Moreover, they must base their decisions on information that is local in time (outdated information is useless [47]) and in place, because information about remote nodes takes a long time to travel.

As shown in Figure 4, a hierarchical scheduler is a tree, such that an interior node is a scheduler and each leaf node is a cluster. Each scheduler controls a domain that consists of either scheduler subdomains or clusters. In its domain, a scheduler is a central resource, so there is no control integrity problem. Allocation of new jobs proceeds as follows. When a job is created in a cluster (marked parent cluster), a scheduler (marked initial scheduler) of a suitable level is selected along the scheduler tree, and it allocates the job to one of its direct subdomains. The scheduler of the latter recursively allocates the job to one of its direct subdomains, until the allocation reaches a cluster (marked child cluster). This target cluster initiates transportation of the job through the point-to-point network. A disadvantage of this hierarchical control system is that borders are created between domains of clusters. Nodes that are close in terms of network distance can have a large allocation distance: consider the nodes on both sides of the border between two subdomains in figure 4. For a scalable system, the information each scheduler has must be limited to a fixed number of global properties of its domain. A good choice of these properties is crucial for performance.
In the macro parallel machine, each scheduler maintains the sum of the work load and the average parallelism of the set of jobs in each of its subdomains. These quantities are global to each domain, and correctly characterise the load. The programming discipline described in section 2 is used to find estimates for the work in each job and its inherent parallelism in case it forks. The sandwich strategy in combination with the grain size control mechanism allows the system to construct an execution profile of applications. The parameter used for grain size control correlates with the computational demand of the corresponding job. During or between runs, this grain size control parameter is collected together with the computational demand and the average parallelism of jobs.

New jobs, considering their work demand and inherent parallelism, are allocated to the subdomain where their allocation causes the shortest finish time of all work in the domain. The selection of the initial scheduler to be consulted is based on a heuristic: the overhead incurred by a job may grow with its computational demand. The maximal distance a job may travel is proportional to its estimated work, and the maximal scheduler level to be consulted follows from this distance.

Simulation studies with the parallel functional benchmark were carried out for evaluation of this algorithm [26, 27], and its performance has been compared with flatly distributed algorithms like the gradient strategy [42]. The hierarchical algorithm for the macro parallel machine performs better for those applications where there is a good correlation between grain size descriptor and work (the gain is between a few and 40%, depending on the application) and usually less good for applications where the correlation is stochastical in nature (the loss rises to 44% for the application “10-queens”). To boost performance, therefore, the application programmer may well go to some extra trouble to define a good grain size descriptor. The introduction of domain borders causes a performance loss of a few per cent on average.
7.2 Intra cluster runtime support system

The coarse grain jobs allocated at a specific cluster are further split into tasks to use all the processors in the cluster. In contrast to jobs, tasks will never be copied since all processors in a cluster have access to the shared memory of the cluster. It is the purpose of the intra cluster runtime support system to allocate memory (stack + heap) for the individual tasks and to schedule them for execution. Both the memory manager and the scheduler take advantage of the tree structure of the divide and conquer applications. The use of the sandwich reduction strategy results in a tree of suspended tasks and a number of independently executing leaf tasks. All tasks refer to data in shared memory, but the data sharing between tasks is strictly regulated: tasks can only share data with their ancestors since the sandwich strategy normalises job/task arguments before sparking them in parallel.

The memory manager provides each task with a private heap. When a task runs out of heap space, it reclaims its garbage locally by running a two-space copying garbage collector on its private heap. The garbage collector does not have to query other tasks since the sandwich strategy guarantees that active tasks do not share any data except for data located in a common (inactive) ancestor, hence, tasks cannot hold global pointers into local heaps of other (active) tasks. This avoids the need for global synchronisation (data locking) and allows the runtime support system to reserve only a small amount for to-space since all tasks can time-share a common to-space. The interior tasks in the tree structure cannot be garbage collected until all children have terminated and linked their heap, which includes a result, to the parent heap. The heaps left over after a join of tasks are scattered throughout the address space of the machine. A special allocation strategy to handle these scattered heaps has been developed to avoid time penalties in the garbage collector [38].

The memory manager is also responsible for allocating a call stack for each task. The dynamically changing size in combination with the suspension/resumption of tasks complicates the space efficient allocation of stacks. Two general solutions are available to minimise memory fragmentation:

- The use of demand paged segments in virtual memory.
- The implementation of a stack as a linked list of call frames in the heap. This may cause a performance loss of up to 41% compared to execution on a true stack [3, 40, 43].

The divide and conquer tree structure, however, can be exploited (again) by allocating a (large) stack per processor as follows: The first task starts with its stack set to the bottom of the processor stack and executes until it executes a sandwich primitive. The task is suspended and the processor continues with a child task whose stack is set just above the top of the stack of the suspend task etc. In essence the processor stack is shared between all tasks allocated to that specific processor as a stack of stacks. When a task terminates it automatically discards its state from the processor stack, so the top most suspended task can resume execution and reuse the released stack space to enlarge its own call stack if necessary. The advantage of this stack-per-processor scheme is that efficient stack based graph reduction is supported, while no space is lost to memory fragmentation inside pages.

The stack-per-processor scheme puts a constraint on the task scheduler since parent tasks are bound to the processor that created them: they are not free to migrate to another processor when one happens to be available, even when their own processor is busy. Examples can be constructed where this binding of parent tasks leads to the loss of practically all parallelism [26], but for practical applications it does not hamper performance. Simulation studies with the application benchmark showed no degradation in performance at all, compared with a regime where parent tasks are allowed to migrate. This might be attributed to the fact that the join parts of parents in the benchmark are responsible for a negligible part of the computation. Therefore
simulations were done with a synthesised benchmark of fork join applications where the join parts consumed the majority of processor cycles. Even this caused no performance difference, except for the group of synthesised applications where the join parts are responsible for almost all of the computation (in this case 93%), and these very improbable applications suffered a degradation of only 5%. In practice the parent binding property is not at all harmful, and the gain offered by its simpler memory management will outweigh incidental losses.

8 Micro parallel reduction machine

In section 6.2 the generation of instruction level parallelism was discussed for programs written in a functional language. As long as the dependencies of the computations are respected, the order of execution of instructions is unconstrained. However, the STORE-operations in section 6.2 can only be executed in parallel if the memory has several ports that can be accessed in parallel. The architecture has to be capable of exploiting this potential parallelism. This observation plays a major role in the design of a special purpose architecture and distinguishes the G-line and G-hinge from other architectures [12, 33, 56, 70] developed to implement lazy functional languages.

Two different VLIW architectures have been designed: the G-line [45] and the G-hinge [46]. The G-line is an abstract architecture exploring the maximum instruction level parallelism possible. The G-line is capable of constructing a subgraph (e.g. a closure) in the time needed for a single memory move operation. The G-line is an idealised architecture because it does not impose limits on the number of memory units etc. The G-hinge uses properties developed in the G-line, but in contrast to the latter is realistically dimensioned, which causes some loss of parallelism.

The G-hinge is specifically designed for graph reduction. This does not preclude the use of standard VLIW techniques, such as parallel ALU and FPU operations. The multiway jump unit used in the NORMA [56] and GRIP [55] architectures for graph reduction can be built into the G-hinge as well.

VLIW machines, such as the G-hinge have a number of functional units that operate synchronously and in parallel. In a VLIW machine for graph reduction it is particularly useful to have units that can access different memory banks in parallel. Unlike memory operations in a vector machine, these operations are irregular. A distribution technique is suitable to make multiple parallel operations possible on the memories that contain the graph and the stack. Distribution of operations means partitioning a data structure and storing the parts in different functional units of the machine. Some data structures are replicated, so that identical copies of the same data structure are kept in different units of the machine. Different parallel operations on this data structure can be done, provided the copies remain identical.

Figure 5 shows that three types of units exist in the G-hinge: heap units, stack units, and compute units. Each unit has a bidirectional connection with each of the global buses. Stack and heap units contain a memory bank, an adder, a copy of the stack pointer or the heap pointer, and logic to select a bus to be accessed. The compute unit contains the data path and program control logic. The program counter addresses the VLIW instruction memory. A single VLIW instruction is subdivided into slots. When a VLIW instruction is addressed, the contents of the slot is loaded and executed in the corresponding unit. The main difference between the G-hinge and other VLIW machines is that the memory units of the G-hinge perform address calculations, whereas normally memory units are passive. Each G-hinge heap unit has a register that maintains the current heap pointer (hp), so that address calculations using the heap pointer can be performed locally within each heap unit. The collection of all heap units with the machine implement a single address space. Similarly, the stack units implement a single address space and each stack unit has a copy of the global stack pointer.
The way distribution techniques are used can be shown using a fragment taken from the code to construct a subgraph of the function append. This is shown in figure 6, with append abbreviated to ap. For the example to be a good illustration of the capabilities of the G-hinge, it has been assumed that the compiler front end is naive in the sense that although the first argument to append is known to be in head normal form after the CASE test (see section Status of the VLIW code generator), it still generates calls to hd and tl. The subgraph to be written in parallel is thus: (: (hd xs) (ap (tl xs) ys)). This takes only two machine cycles. The cells in Figure 6 are arranged in an unusual way to show that the two stack units are read during the first cycle, so that all 13 heap fields involved can be constructed during the second cycle.

Before the subgraph can be written, the stack must be read out to deliver the pointers xs and ys. This takes one machine cycle, as the two stack units can operate in parallel. After being read from the stack, each pointer is placed on a separate global bus, which is designated by the compiler. The two stack units thus obey similar instructions, that differ only in the output bus number. The second and last machine cycle also feeds each of the heap units with an appropriate slot of the VLIW instruction, so that each unit knows for which part of the subgraph it is responsible. The units 0,3,4,6,7,10 and 11 write immediate data contained in the instruction slot to the appropriate heap field. Units 5,9 and 12 copy the value from the correct bus into the fields they are responsible for. Unit 1 must store the value a, which it computes by adding 3 to the current heap pointer. Similarly units 2 and 8 add an immediate constant to the value of the heap pointer. Since each heap unit has a copy of the heap pointer, these operations do not restrict the parallelism.

The entire subgraph can thus be created in two machine cycles: one to fetch operands from the stack and the next to write the subgraph. For graphs that require more fields than there are
heap units, the compiler arranges the code such that the subgraph is split into several smaller parts that do fit the machine. This lowers the parallelism but raises the cost effectiveness of the machine. Reading the stack may cause similar difficulties, when more than one stack item has to be read from the same unit.

Simulations are being performed to determine sensible values for the machine parameters, such as the number of buses, stack, heap and compute units. Using a benchmark of three programs we found [45] that with 4 stack units and 4 global buses the maximum parallelism for graph operations is not affected. The maximum number of heap operations that can be gathered in a single VLIW instruction is equal to the maximum number of fields in a subgraph that can be constructed at once. In some cases this may be more than 100 instructions, but this is rare. 12 units are sufficient to create 69% of all subgraphs in one machine cycle, between 12 and 24 units are required for the next 25%. 95% of all the subgraphs can be created with a machine that has 24 heap units. If scheduling of instructions on the heap is integrated with VLIW scheduling of the whole program (including the more sequential parts of the program) we expect that a total of 8 heap units will be enough to exploit the parallelism possible.

The basic operations of the compute and memory units are movements of bit fields. The compute and memory units can be programmed to combine bit fields in any way desired. The G-hinge architecture is thus capable of building an arbitrary data structure, and not just tagged binary application nodes. Given a suitable compiler, the G-hinge architecture is thus suitable to run other execution models like the G-machine [30], the Spineless Tagless Machine [53] and the \( (v, G) \) machine [3]. In the latter case no special stack units are needed.

9 Conclusions

The paper highlights major results of an extensive study into five essential components of a parallel reduction machine: a methodology to write parallel functional programs, sophisticated compilers and code generators, efficient run time support matching the programming methodology and an architecture with a VLIW processor designed for graph reduction. The results of these studies, in particular measurements, have been obtained by careful simulation, using advanced tools for architecture simulation [48]. The major achievements are summarised as follows:

1. Programming methodology for developing parallel functional programs.

   The methodology is based on explicit annotations and program transformations. It has been successfully applied to a number of algorithms resulting in a benchmark of small and medium size parallel functional programs. Speed-up figures have been measured ranging from 2.2 to 4.5 with an overall processor utilisation of at least 50%.

   The method to implement I/O in a functional language is capable of dealing with a complicated interactive system as the Macintosh user interface. Parts of this interface have been implemented. The most important properties of referential transparency are retained, thus enabling correctness proofs to be made as in a pure system.

2. Compilation techniques for functional languages.

   An automatic tool has been developed for transforming untyped SASL-programs into efficient, strong polymorphically typed versions. The typed programs resulting from this transformation run about half the speed of directly hand coded versions.

   A compiler (FAST) has been designed and implemented to perform strictness, boxing and other analysis on non-flat domains.
3. **Code generation techniques for RISC and VLIW architectures.**

The intermediate code produced by FAST is transformed into C (for portability) by a sophisticated back end (the FCG code generator) that performs various optimisations. The speed of programs compiled by the FAST compiler and FCG code generator is slightly better than when compiled by other state-of-the-art compilers.

The Stoffel code generator targets VLIW architectures. Functional languages offer good opportunities to allocate registers and to schedule instructions in a VLIW architecture.

4. **Systems architecture.**

A VLIW graph reduction processor has been designed to be used as processing element in the shared memory clusters of our machine. The design, called G-hinge, allows the construction of a suspension in one machine cycle. As a consequence the machine is capable of running lazy-code almost as fast as strict code. Where compiler analysis fails to detect strict arguments the G-hinge still catches up to provide satisfactory speed.

5. **Efficient runtime support.**

For the macro parallel machine a hierarchical distributed scheduler has been designed that takes advantage of grain size information of parallel jobs. This information is freely available for programs developed with the annotation/transformation methodology. For a benchmark of small to medium size programs, performance of this scheduler is up to 40% better than known flat distribution algorithms (like the gradient method).

Scheduling tasks within a shared memory cluster also uses a special algorithm that only allocates a single stack per processor. All tasks on one processor use the same stack. Simulation with the benchmark indicates that this algorithm offers possibilities for efficient implementation with little loss of parallelism (less than 5%).

Most results described above have been used to build a working prototype of a single cluster machine. At present the machine is based on four 88000 processors that share a 64 Mbyte memory. In future the machine will be extended with more clusters and a point-to-point network. Eventually the 88000 processors will be replaced by G-hinge processors.

**Acknowledgements**

We thank Henk Muller and the two referees for their comments on a draft version of the paper. The support of the European Institute for Technology under grant no. I88 is gratefully acknowledged. The FAST compiler represents joint work with Hugh Glaser and John Wild, which is supported by the Science and Engineering Research Council, UK, under grant No. GR/F 35081, FAST: Functional programming for ArrayS of Transputers.

**References**


Biographies

Henk P. Barendregt received the Master’s degree and the Ph.D. degree in Mathematics from the University of Utrecht in 1986 and 1971 respectively. In 1981 his book on the Lambda Calculus was published, which has been translated into Russian and Chinese. Until 1986 he was a Senior Lecturer at University of Utrecht in Philosophy and Foundations of Mathematics. Since 1986 he is a full professor of Foundations of Computer science at the Catholic University of Nijmegen. His research interests are the foundations of Mathematics and Computer Science.

Marcel Beemster received the Master’s degree in Computer Science in 1986 from the University of Amsterdam. For his Master’s thesis he studied position detection and system design for an autonomous cart. To 1990 he participated in PRISMA, a joint Dutch project that built a 100 node parallel database machine, in which he worked on design and implementation of compilers for the language POOL (Parallel Object Oriented Language). He is currently a Ph.D. student working on the compilation of lazy functional languages to VLIW-architectures.

Pieter H. Hartel received the Master’s degree in Mathematics and Computer Science from the Free University of Amsterdam in 1978 and the Ph.D. degree in Computer Science from the University of Amsterdam in 1989. He has worked at CERN in Geneva, the University of Nijmegen (The Netherlands) and the University of Southampton (England). He is currently a Senior Lecturer at the University of Amsterdam, in the Department of Computer Systems. His research interests are in the design of compilers, operating systems and architectures for functional languages.

Louis O. Hertzberger received the Master’s degree in experimental physics in 1969 and the Ph.D. in 1975, both from the University of Amsterdam. From 1969 till 1983 he was a staff member in the High Energy Physics group, later the NIKHEF-H (Dutch Institute for Nuclear and High Energy Physics). In 1983 he was appointed as professor in Computer Science. His current research interests are in the field of parallel computing, intelligent autonomous robotics and their application in industrial automation.

Rutger F. H. Hofman graduated in physics and computer science in 1987 in Utrecht. Until 1992, he worked on multiprocessor heuristics for the Dutch parallel reduction machine, for which he soon hopes to receive the Ph.D. degree. He currently works at the Free University in the Orca project. His research interests include parallel applications and run-time support systems.

Koen G. Langendoen received the Masters degree in Computer Science from the Free University of Amsterdam in 1988 and hopes to receive the Ph.D. degree in Computer Science from the University of Amsterdam in April 1993. He has worked at the University of Amsterdam and is currently a Senior Researcher at the Free University of Amsterdam, in the Department of Computer Science. His research interests are the design of compilers, operating systems, and architectures for parallel programming languages.
Liang-liang Li received the B.Sc, M.Sc and Ph.D. degrees in Computer Science from the Changsha Institute of Technology, China, in 1982, 1984 and 1988, respectively. He became a lecturer in Changsha Institute of Technology in 1988. From November 1988 to March 1991 he has been with the University of Amsterdam. In March 1991, he joined ECRC in Munich, where he is currently involved in the research on an or-parallel constraint logic programming system. His research interests include the design of compilers, operating systems and architectures for declarative languages.

Robert Milikowski received the Master’s degree in Physics in 1971 and the Master’s degree in Mathematics and Computer Science in 1987, both from the University of Amsterdam. He has worked on science policy as an aid to members of the Dutch parliament between 1971 and 1983. He is currently a researcher at the University of Amsterdam, in the Department of Computer Systems. His research interests are in the design of VLIW architectures.

J. C. Mulder received the Master’s degree in Mathematics from the University of Utrecht in 1983 and the Ph.D. degree in Computer Science from the University of Amsterdam in 1990. He has worked at the University of Utrecht, the Centre for Mathematics and Computer Science in Amsterdam and the University of Amsterdam. Currently he holds a research position at the Catholic University in Nijmegen in the department of Theoretical Computer Science. His research interests are mathematical logic and the design of declarative languages.

Willem G. Vree studied applied physics in Amsterdam where he obtained the Master’s degree in 1973. He worked for 5 years on pattern recognition at CERN in Geneva. Next his interest shifted to distributed real time systems at the Dutch Waterboard. Later he joined the University of Amsterdam and completed a Ph.D. thesis on parallel reduction machines in 1990. Currently he is head of strategic research in information technology at the Dutch Waterboard. His research interest is in declarative systems and their applications.