Graph Reduction on Shared-Memory Multiprocessors

Koen Langendoen

Contents

Acknowledgements

This research covered in this book was performed during my employment at the University of Amsterdam as member of the EIT Reduction Machine project. I like to thank Bob Hertzberger and Wim Vree for providing me the opportunity to design and implement the runtime support for parallel graph reduction on shared memory multiprocessors.

Special thanks go to the co-authors and proof-readers of various papers written at the UvA: Dirk-Jan Agterkamp, Marcel Beemster, Pieter Hartel, Rutger Hofman, and Henk Muller. A large part of this book is founded on ideas and results of collaborative work presented in those papers. The following people gave useful feed-back on their work as discussed in the survey on parallel implementations of lazy functional programming languages in Chapter 3: Thomas Johnsson ($<\nu,G>$), Luc Maranget (GAML), Herbert Kuchen (PAM), David Lester (HDG), Marco Kesseler (PABC), and Simon Peyton Jones (GRIP). Frans Kaashoek and Henri Bal did an expert job in spotting functional jargon throughout the text, and prompted me to clean up the chapter about the ins and outs of functional programming and its implementation.

The "vrijdag club" lead by Pieter Hartel has proven to be of great value for trying out conference talks and new ideas. I thank all members for their interest and critical response on my research activities. Finally I express my sincere gratitude to everybody involved in the MiG project: Henri Groeneweg (assembly hacking), Pieter Hartel (compiler support), Rutger Hofman (application tuning), and Henk Muller (debugging expert). Your collaboration, support, and advise when implementing the WYBERT system made it possible to include the necessary hard numbers supporting the efficiency of the design.

Chapter 1

Introduction

Technological improvements have produced sequential processors that run thousand times faster than a few decades ago, but still many scientific and engineering problems cannot be solved on today's fastest supercomputer: 3 dimensional fluid dynamic models, non-linear finite element computations, real-time video processing, etc. Other large problems, like weather forecasting and blood flow analysis, can only be solved by simulation and approximation. This involves huge amounts of computation to obtain acceptable results; for better results, more computation is required.

The ever increasing demand for processing power is the driving force for the development of parallel processing. Although the speed of processors has been steadily increasing, it roughly doubles every four years, engineering constraints and physical effects such as the finite speed of light make it more difficult to speed up the fastest processors through hardware technology improvements alone. In addition, the important development of Very Large Scale Integration (VLSI) technology, which allows tens to hundreds of thousands of transistors on one single chip, has reversed "Grosh's law" that states that the most powerful uniprocessor has the best price/performance ratio [Ein-Dor85]. A parallel computer built out of a collection of small processors is more cost effective than an uniprocessor system since the pay-off for increased construction effort on a silicon chip is less than linear.

1.1 Parallel computers

A parallel computer consists of a (large) number of processing elements that co-operate to solve a single problem. The actual construction of parallel computers was already started around 1970 with pioneering projects like the ILLIAC IV at NASA and C.mmp from Carnegie-Mellon University. A wide variety of parallel computers has been designed and implemented since.

An important characteristic of parallel computers is how control is organised in the machine: either each processor executes its own program, or all processors receive the same instructions from a central source. These two possibilities are named MIMD and SIMD, respectively, from a classification made by Flynn [Flynn72], see Table 1.1. The conventional "von Neumann" processor is viewed as a Single Instruction stream operating on a Single Data stream (SISD). A first step towards parallel computing consists of introducing Multiple Data streams (SIMD), and a second step adds Multiple Instruction streams (MIMD).

	Single Data	Multiple Data
Single Instruction	SISD (von Neumann)	SIMD (vector, array)
Multiple Instruction	MISD	MIMD (multiprocessors)

Table 1.1: Flynn's taxonomy of computer architectures.

In SIMD computers, also known as array or vector computers, all processors simultaneously apply the same instruction to different (local) data. This type of machine is suitable for applications like image processing that perform the same operation on many data values (e.g., inverting all pixels of a picture). These machines are built out of simple, but numerous, processing elements; for example, the Connection Machine CM-1 can contain up to 65,536 (2^{16}) 1-bit processors.

MIMD is seen as a more general approach since the individual control of each processor allows for the exploitation of irregular parallelism. Most research is concentrated on MIMD parallel computers, as is the remainder of this book.

MIMD parallel computers can be divided into two categories based on whether or not physically shared memory is included in the architecture. In shared-memory *multiprocessors* each processor can access all memory cells with ordinary read/write instructions, which move data from/to a location in the global address space. In distributed-memory *multicomputers* each processor can only access a part of the memory directly, and needs some other way to access 'remote' data, for example, message passing. Figure 1.1 illustrates the basic machine architecture of both categories.

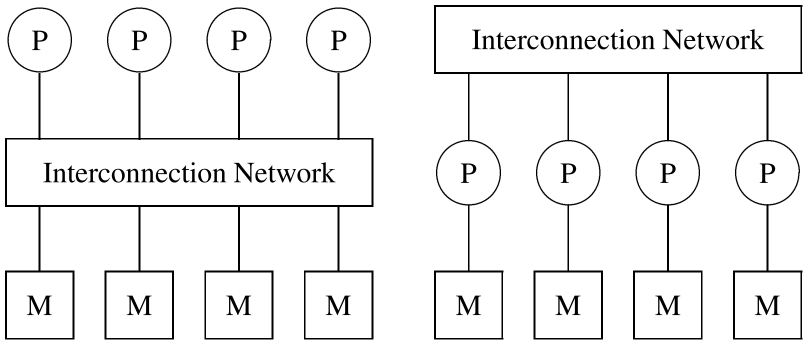


Figure 1.1: Shared- and distributed-memory parallel machines.

1.1.1 Shared-memory multiprocessors

The interconnection network that links processors and memories together is the key factor in the overall performance of a multiprocessor. In the simplest case all processors are connected to a single bus, along with a memory module. When reading or writing data, a processor issues a normal memory request on the bus; during the bus transaction other processors that want to access memory are blocked until the current transaction has finished. This time sharing of the single bus severely limits the performance: with a few processors, say 8, the bus will be completely saturated. Therefore each processor is usually given a cache as shown in Figure 1.2.

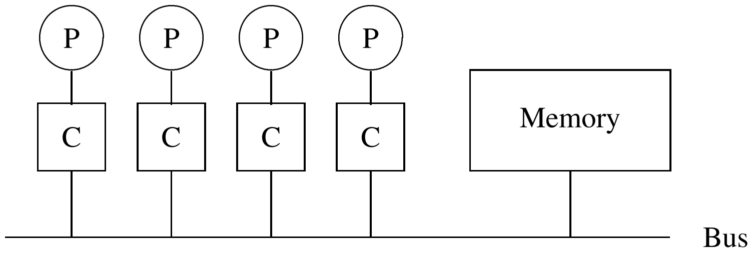


Figure 1.2: a multiprocessor with a single bus and caches.

The caches maintain copies of frequently accessed data values so that most memory requests of a processor can be satisfied locally without accessing the system bus. This effectively reduces the traffic on the single bus, so the number of processors can be increased, say up to 32, before the bus saturates. Caches introduce a memory consistency problem where caches contain different values

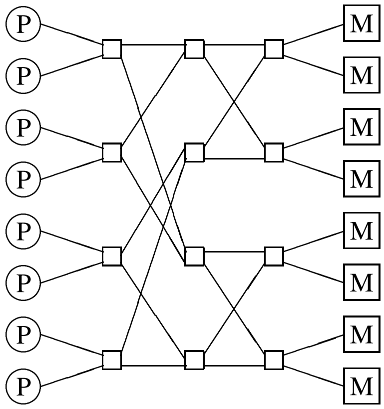


Figure 1.3: A multiprocessor with an omega network.

for the same memory location. For example, if both processors 1 and 2 read the contents of memory location m into their cache, and processor 1 modifies it subsequently, then the next read of location m by processor 2 gets the old *stale* value. This *cache coherence* problem can be solved, for example, by having the caches constantly *snoop* (i.e. monitor) the bus. Whenever a read or write is observed of a location that has been cached locally, the cache takes an appropriate action like invalidating the cache entry [Sweazy86, Stenström90]. Such solutions, however, require additional hardware or decrease performance, or both.

Although caches alleviate bus saturation somewhat, large multiprocessors require a more sophisticated processor-memory interconnection network that provides parallel access paths to memory. The omega network in Figure 1.3 is the prime example of an interconnection switch where several processors can access different memory modules in parallel; in the best case the omega network can service all processors simultaneously. However, if multiple processors issue a reference to the same memory module, contention in the network arises and delays are incurred; in the worst case the performance of the omega network is even lower than that of a single bus because of the multiple switching delays. This problem, known as “hot spot contention” [Pfister85], has been addressed by the NY Ultracomputer project [Gottlieb83]: a network switch combines multiple references to the same memory location into one request. This reduces network congestion, but at the expense of increased hardware complexity.

A fundamental problem of all multistage switching networks is that connecting n processors to n memory modules (as $n/2$ rows of $\log_2 n$ switches), requires $n/2 * \log_2 n$ switches. Furthermore each memory reference has to cross $2 * \log_2 n$ switches. This makes it difficult and expensive to build multiprocessors with large number of processors; for example, a 1024 node machine requires 5,120 switching elements.

1.1.2 Distributed-memory multicomputers

In contrast to multiprocessors, distributed-memory multicomputers are straightforward to build since each processor-memory pair is more or less independent of the others. The processing elements communicate via message passing, which causes the interconnection hardware to be of relatively low significance for system performance: data is efficiently transported in large chunks, as opposed to individual words in case of multiprocessors, and typical applications are programmed as a set of coarse grained tasks so context switches to hide transport delays can be tolerated. Furthermore the large software overhead to handle a message is larger than the actual transmission time in the network. Therefore a simple bus-like interconnection network, such as a local area ethernet, provides already enough capacity to construct large multicomputers (up to ca. 100 nodes). Of course, the single bus will inevitably become a bottleneck when increasing the number of processing elements or decreasing the application's grainsize, so many multicomputers have been built already with a collection of point-to-point connections; Figure 1.4 shows the popular (Transputer) grid and hypercube networks.

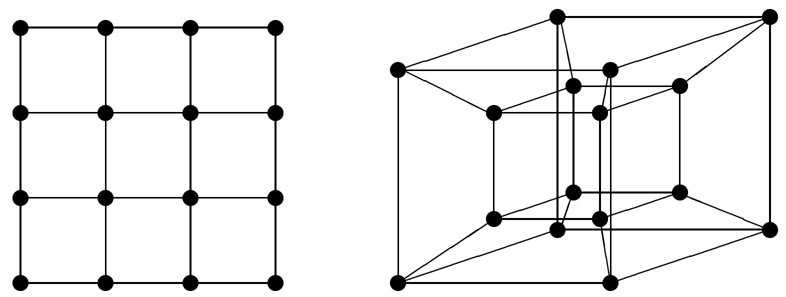


Figure 1.4: Grid and hypercube multicomputers.

A grid is less difficult to program than a hypercube and can be packed more densely since it has only four external connections. The hypercube has the advantage that the network diameter (i.e. the maximum distance between a processor and any other) grows logarithmically with the number of processors. This advantage, however, is rather small since measurements on advanced networks have shown that the message passing overhead in software makes all communication costs equal; it is virtually as expensive to communicate with a neighbour as with a processor at the other side of the network [Bokhari92].

1.2 Parallel programming

The introduction of parallel computers has aggravated the software crisis that already existed for sequential computers. The programmer not only has to devise a suitable algorithm to solve a particular problem, but the additional complexity of synchronisation and communication between co-operating processors has to be mastered as well to take advantage of the computing power offered by parallel computers. Efforts to hide parallel computers from the user by having the compiler automatically extract the parallelism from an application have not been proved successful for general MIMD computers.

Two basic approaches to writing parallel software can be distinguished: the shared data model that matches with the shared-memory multiprocessors, and the message passing model that matches with the distributed-memory multicomputers. Both approaches require the user to explicitly handle parallelism in the application program.

1.2.1 Shared data; the evolution

Although difficult to build, shared memory multiprocessors are a commercial success since they are fairly easy to program. All processes can access the whole shared memory, so sharing of data structures and variables is straightforward. This allows for easy and flexible communication between processors since when one process updates some variable and another reads it afterwards, the underlying hardware automatically returns the value just stored. The programmer only has to be concerned with synchronising the activities of the co-operating processors to avoid inconsistencies. Fortunately, many standard synchronisation techniques developed for sequential multi-tasking computers, like semaphores and monitors, can be applied directly for programming multiprocessors.

The powerful shared-data model facilitates an easy evolution of sequential code to parallel code by means of well-understood principles of managing concurrency. For example, many dusty-deck FORTRAN applications can be parallelised by changing **for**-loops into **forall**-loops where the iterations are processed concurrently by different processors.

1.2.2 Message passing; the revolution

The lack of shared data in distributed-memory multicomputers prevents the re-use of old software, and forces programmers to essentially develop new applications from scratch. The difficulties are that data has to be explicitly distributed over the memory modules in the multicomputer, and that the communication between co-operating processors has to be done by hand through sending/receiving messages.

In the basic case, the programmer is provided with low-level **send** and **receive** primitives to communicate with direct neighbours as, for example, in Occam on Transputer systems. Although these primitives suffice to program a multicomputer, the programmer is bothered with difficult issues like the buffering and routing of messages. Therefore several software packages (e.g., EXPRESS and CSTools [Hellberg92]) have been developed that take care of the nitty-gritty details of message passing, and provide the user with communication between arbitrary processes in the multicomputer.

The *remote procedure call* mechanism [Birrell84] abstracts even further from the hardware by making communication look like an ordinary procedure call. Instead of executing the procedure locally, a *stub* routine gathers the parameters in a message, and sends it to the remote processor; another stub routine unpacks the message and performs the actual procedure call, finally the reply is transported back to the original processor through both stubs. Although this scheme hides the message passing from the user, it is usually not completely transparent: for example, passing pointers as parameters is often forbidden, hence the user has to know whether a procedure is invoked locally or remotely.

To achieve good performance on distributed-memory multicomputers, the programmer has to carefully distribute code and data over the processor-memory pairs so that communication requirements are minimalised. Unfortunately, few tools exist to assist the programmer in this task.

1.2.3 Distributed shared memory; best of both?

Various researchers have proposed to combine the programming ease of shared-memory with the construction ease of distributed-memory by simulating a global shared address space on top of a multicomputer. In these *distributed shared-memory* machines, a process on any processor can access memory anywhere in the system through ordinary read/write instructions, but while references to local memory are satisfied immediately, references to data located on another processing element are intercepted and incur considerable delay since messages have to be exchanged with the owning processing element to access the remote data. The handling of remote references is done transparently by low level system software, so the user is presented with the impression of shared memory on a distributed-memory multicomputer. Because of the difference in access time of local and remote data, distributed shared-memory machines are also known as NUMA (Non-Uniform Memory Access) machines

Distributed shared-memory machines differ largely in the way remote references are handled, and in the granularity of access to shared data. The Cm* [Swan77] represents one end of the spectrum: each memory reference is checked in the micro code of the MMU, and messages carrying one data word are sent over a backplane bus in case of remote references. Remote references take about ten times as long as local ones, and the programmer is solely responsible for achieving good performance by placing code and data appropriately in the machine. The page-based scheme of Li and Hudak [Li89] is at the other extreme and uses standard virtual memory techniques. The global address space is partitioned into fixed size pages, which are distributed over the processing elements in the system. A reference to a non-resident page causes the hardware to generate a page fault as usual, but now the operating system fetches the page from the owning processing element, instead of from disk. To reduce thrashing, read-only pages may be replicated at many processing elements, but writable pages must reside at one processing element for consistency.

A disadvantage of the above hardware based schemes is that hardly ever the right amount of data is transported: one word is too small, and a page (say 4Kbyte) is usually too large. Therefore intermediate designs like Orca [Bal90] and Linda [Carriero89] have been proposed that support the sharing of variable sized objects. The programmer has to define these shared data objects, and controls the granularity of sharing. Now the compiler provides the illusion of shared memory by generating special code to access shared objects. For example, shared objects in Orca are replicated, and updates are compiled into broadcast messages to keep the copies consistent.

It remains to be seen whether or not the distributed shared memory paradigm will defeat the raw message passing model for programming large scale distributed memory multicomputers. The simple familiar shared-memory programming model is a definite win, but the performance is still a weak point. Although a high-level distributed shared memory application will probably never match the performance of a hand-coded message-passing equivalent, the rapid improvement in efficiency indicates that distributed shared memory implementations will reach acceptable performance for all but the most time-critical applications in the near future.

1.3 Functional programming

A grand challenge for computer scientists is to domesticate the power of parallel computers by providing a suitable high-level programming environment that hides the nasty details to the ordinary user (i.e. application programmer). Ideally a programmer has only to conceive a parallel algorithm, which will be automatically compiled and executed on the specific parallel computer at hand. Of course this automatic high-level approach looses on execution efficiency in comparison to hand-crafted low-level programming, but history has shown that performance costs are often less important than the ease (i.e. productivity) of programming. For example, assembly programming is nowadays considered to be a necessary evil to squeeze out the last drop of performance; in 99% of all cases the high-level language compiler does its job good enough to satisfy the average user. From a user's perspective the ideal parallel programming language should:

- offer a high-level of abstraction to master software complexity.
- support the shared-memory parallel programming view, which is much closer to sequential programming than message-based parallel programming.
- hide low-level issues like communication and process synchronisation.
- be easy to reason about, i.e. have clear semantics, so programs can be proven correct.
- run transparently on a range of hardware configurations (portability).
- perform reasonably efficient.

Modern functional programming languages meet many items on this list of requirements because of a number of essential features that will be discussed below.

First of all, functional languages provide means to program at a high level of abstraction so the programmer does not have to take lots of technical details into account. For example, a programmer may allocate huge numbers of data structures without ever releasing storage of structures that are no longer in use; the underlying garbage collector automatically reclaims unused memory space and thus frees the programmer from the finite memory constraint. As another example, the support of higher-order functions allows the programmer to treat functions as ordinary values although the bare hardware only supports simple data types like integers and floating point numbers.

Higher-order functions and lazy evaluation, which will be studied in depth in Chapter 2, are two concepts that improve the modularisation of software since they can be used to 'glue' program components together in ways that are not supported by modular imperative languages like Modula-2 and Ada, as argued in [Hughes89]. This is an important advantage of functional programming languages since writing well-structured modularised programs is the key to software engineering. In addition, the strong polymorphic typing of many functional languages reduces the development effort of software as well.

Secondly, functional languages are well known for their *referential transparency*: a particular expression always denotes the same value independent of the context where it is evaluated. This is a consequence of the lack of assignment that prohibits the expression evaluation to have side-effects. This not only avoids a major source of programming errors, but it makes functional programs much easier to reason about for humans as well as programs. In particular compilers benefit from the lack of destructive updates since that greatly simplifies the data dependency analysis.

The lack of destructive updates guarantees that any set of expressions can be computed in parallel without destroying the correctness of the programs' result. It is, of course, not beneficial to execute an expression in parallel if the amount of computation does not outweigh the overhead costs. Unfortunately it is too difficult for a compiler to work out the grainsize of an arbitrary expression, so the user has to denote which expressions are worthwhile to be evaluated in parallel. This is, however, all that is required to run a functional program on a parallel computer: because of the clean semantics the runtime support system can take care of low-level issues like load-balancing, communication, and process synchronisation without further assistance of the user.

The referential transparency of functional programming languages facilitates a simple parallel programming environment where the user only has to place a few annotations in the source code to obtain parallel execution; the low-level details of parallel programming are hidden from the user through the accompanying runtime support system. This approach retains all the advantages of sequential functional programming (e.g., high-level abstractions, ease of reasoning, and expressive power) and is highly portable since only the runtime system has to be ported when upgrading to a new machine.

Despite all advantages listed above, functional programming languages have not (yet) been generally employed to program parallel computers because of several reasons. Until recently, the lack of fast implementations on conventional processors has hampered acceptance, but now state-of-the-art functional language compilers generate object code whose quality approaches that of standard compilers for imperative languages. For example, measurements in [Smetsers91] show that often execution times are within a factor three of comparable C programs, but the memory usage is still orders of magnitude higher. The latter is caused by the lack of destructive updates, which forces data structures (arrays) to be copied to create new versions even when only one element has to be modified. A lot of ongoing research is directed at developing methods to detect at compile-time whether or not an update can be done in-place. Two promising approaches are the usage of monads [Wadler90] and the concept of unique types [Smetsers93].

A more serious and fundamental problem is the drawback of referential transparency: it is impossible to express non-determinacy in (pure) functional languages. This severely limits the interactive usage, for example, how should keyboard interrupts be modeled in a functional operating system? Debugging is another open problem: lazy semantics imply a control flow that bears little resemblance to the logical structure of a program, hence, imperative debugging strategies like tracing and break-pointing are of no great use.

1.4 The EIT Reduction Machine project†

The EIT Reduction Machine project is a joint effort of the University of Amsterdam (UvA) and the University of Nijmegen (KUN) aimed at the development of an efficient functional language implementation on large scalable parallel computers. The project tackles some of the fundamental and practical problems of functional languages on parallel machines as outlined above, and builds on the experience gained with the APERM prototype of the Dutch Parallel Reduction Machine project [Hertzberger89, Barendregt87].

The APERM machine was designed to study the feasibility of parallel reduction machines. Applications are programmed in a lazy functional language augmented with an annotation to denote divide-and-conquer parallelism.

†Supported by the European Institute of Technology, grant nr EIT I-88.

These applications are automatically scheduled for execution on the APERM architecture that consists of a number of processor local-memory pairs interconnected through dual ported memories for high-speed data communication. Measurements showed that the processor to memory connection in the APERM prototype was heavily under utilised: typically less than 10% usage [Hartel88b]. Therefore the HyperMachine, as shown in Figure 1.5, includes shared-memory multiprocessors to improve the price/performance ratio of the hardware.

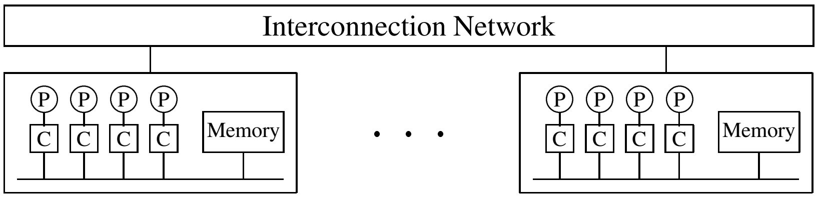


Figure 1.5: HyperM architecture.

The HYbrid Parallel Experimental Reduction Machine (HyperM), which forms the back-bone of the EIT project, is thus the successor of the APERM prototype and comprises both shared- and distributed-memory. At the top-level, HyperM is configured as a distributed-memory machine: a number of clusters with local memory interconnected by a high speed network; at the bottom level, each cluster itself is a multiprocessor that consists of a few processors connected to a shared memory.

Applications for the HyperMachine are programmed in a standard functional language that has been augmented with a single primitive to denote parallelism. This primitive is called the *sandwich* annotation [Vree89] and is based on the divide-and-conquer paradigm. The user (recursively) divides the initial problem into independent sub tasks that can be solved in parallel. The compiler and runtime support of HyperM take care of distributing, scheduling, and synchronising the divide-and-conquer tasks across the machine. A large class of problems can be expressed directly as divide-and-conquer programs, while transformational methods have been developed to handle synchronous process networks as well [Vree90].

The runtime support system (RTS) of HyperM is responsible for exploiting the two-level memory hierarchy of the machine in a transparent way; the programmer regards HyperM as a divide-and-conquer machine with a single global data store. To simplify the complex resource management task, the RTS is split into two parts: the inter cluster RTS that distributes tasks over clusters and handles communication on the interconnection network, and the intra cluster RTS that handles task scheduling and storage management inside one shared-memory cluster. To minimise interaction between the two parts, tasks are classified into two categories: large *jobs* that may be allocated at any cluster in the HyperMachine, and small *threads* that are limited to one cluster. The RTS classifies each task based on a grainsize measure that is provided by the programmer as part of the sandwich annotation; a single threshold value suffices to distinguish jobs and threads.

1.5 WYBERT

The research objective addressed in this book is to show that it is possible to efficiently implement functional languages on parallel machines, which is reflected in the acronym WYBERT that stands for "Would You Believe Efficient Reduction Today?". The question is not answered in a general setting, but in the context of the EIT Reduction Machine project: WYBERT is the name of the intra cluster runtime support system of the HyperMachine. Because of the clear separation of concerns between the *inter* and *intra* cluster runtime support system, WYBERT is essentially the runtime support system for a functional language implementation on shared-memory multiprocessors based on the divide-and-conquer paradigm.

The WYBERT system takes advantage of the regular parallel structure of annotated divide-and-conquer applications that unfold into a set of independent sub problems. These logically independent sub problems, however, can share expressions at the graph reduction level. To avoid inconsistencies, parallel graph reduction systems usually equip graph nodes with locks to enforce mutual exclusive access. In contrast, WYBERT adopts the APERM approach of evaluating shared redexes in advance, which eliminates the existence of shared writable data. The so called *sandwich* reduction strategy was originally developed for distributed memory systems and has been adjusted to match the different requirements for execution on shared-memory multiprocessors. Several transformations have been developed to overcome the effects of the additional eager reduction, but these transformations are only needed in exceptional cases.

Forcing the parallel tasks to be independent at the graph reduction level has a number of advantages:

- Graph reduction can proceed without locking data that resides in shared memory. This can boost performance for certain applications in comparison to general parallel implementations of functional languages that do require mutual exclusive access of some parts of the shared data, as will be discussed in Chapter 6. The performance advantage increases for multiprocessors with larger numbers of processors.

- Each parallel task can garbage collect its own part of the heap without global synchronisations since the lack of writable shared data prohibits the implicit exchange of pointers between parallel tasks. For efficiency, a novel storage management scheme allocates private heaps of parallel tasks such that each task can run an ordinary sequential two-space copying garbage collector. An additional advantage is that by limiting the maximum task size, the runtime support system can use a small time-shared buffer as to-space instead of reserving half of the available memory needed by a traditional global two-space copying garbage collector.
- Stacks of parallel tasks can be efficiently allocated on a single stack per processor to avoid the burden of managing an unknown number of variable-sized stacks. The special WYBERT scheduler takes care of the extra constraint that a runnable task in the stack may not execute until all tasks above of it have finished.

Considerable effort has been put in the development of a working implementation to measure the effects of these advantages of the WYBERT approach. This work includes the development of a new code generator.

as part of a functional language compiler that generates efficient code containing handles for the WYBERT runtime support system.

The main contributions of this book are the design and implementation of a system that supports the efficient parallel execution of functional programming languages on shared-memory systems. The key to success is the *sandwich* divide-and-conquer primitive that produces independent tasks in a shared memory environment. The WYBERT design exploits this fine property by supporting lock-free graph reduction and including two new storage management optimisations for efficient allocation of stack and heap space. Measurements on a four node shared memory multiprocessor show that WYBERT outperforms the common spark-and-wait parallel implementation technique of lazy functional languages.

1.6 Outline

Chapter 2 provides an introduction to functional programming and its main implementation technique: graph reduction. The fundamental concepts of functional programming, higher order functions and lazy evaluation, are studied in considerable detail since these are the key concepts that complicate the efficient implementation.

Chapter 3 gives a comprehensive survey of parallel implementations of lazy functional languages. It discusses fundamental issues raised when extending sequential graph reduction to parallel machines: the need to support a global address space, generation and control of parallelism, and resource control. This discussion provides the basis for the comparison of nine recent designs and prototype implementations of parallel functional language implementations that concludes the chapter.

In Chapter 4 the design of the WYBERT approach to parallel graph reduction on shared memory is presented. It discusses the impact of the divide-and-conquer primitive that reduces some expressions in advance to eliminate the existence of shared writable data. The resulting independent task structure is exploited in the storage management of WYBERT. Two new algorithms for efficient allocation of stack and heap space are discussed in detail including some performance effects obtained by simulation.

Chapter 5 discusses the FCG code generator for the FAST compiler front end [Hartel91a] that 'knows' about the operational semantics of the sandwich primitive and supports compacting garbage collection schemes. The FAST/FCG compiler generates quality code that compares well to other compilers for functional languages. The compiler, in combination with a set of library routines that implement the runtime support system of WYBERT (i.e. task scheduling and storage management), has been used to experiment on a prototype implementation of the HyperM architecture.

Chapter 6 describes the performance of WYBERT as measured using one Motorola *HYPERmodule* that consists of four MC88000 processors equipped with caches connected to 64Mbytes of shared memory. The performance of WYBERT is compared to the standard parallel implementation technique based on spark-and-wait annotations. In addition detailed performance graphs, as produced by a separate monitoring tool, are analysed to show where individual applications spend their execution time.

The book concludes with Chapter 7 that summarises the most important aspects of the WYBERT design (Chapter 4) in relation to the measured implementation results (Chapter 6).

Chapter 2

Functional programming and its implementation

Functional programming languages are referentially transparent, which gives them simple semantics. Since the original development of Lisp by McCarthy [McCarthy60], the design of functional programming languages has focused on increasing the expressive power of the functional model, while preserving the simple semantics. Modern functional languages like LML, Miranda, and Haskell are three examples of this trend; in addition to the basic features of (pure) Lisp, they provide higher order functions, lazy evaluation, abstract datatypes, equations/pattern matching, and static polymorphic type-checking. This chapter gives a short introduction to functional programming and briefly discusses the issues that are relevant for the parallel implementation as will be described in Chapter 3. The reader is referred to [Hudak89] for a detailed survey. Higher order functions and lazy evaluation are studied in depth since these features require special implementation techniques like graph reduction to be executed efficiently on stock hardware. Elaborated discussions of the functional programming style and reasoning can be found in standard textbooks like Bird and Wadler [Bird88] or Field and Harrison [Field88]. All example programs are given in the Miranda† programming language [Turner85, Turner90].

Functional programming languages are founded on the sound mathematical basis of the lambda calculus [Barendregt84]. Throughout their evolution, functional languages have kept with pure mathematical principles without any compromise. This has resulted in a pure declarative style of programming,

†Miranda is a trademark of Research software Ltd.

with emphasis on what to compute, and not on how to compute it. For example, the following Miranda definition of the factorial function

```
fac n = 1                , if n=0
      = n * fac(n-1)     , if n>0
```

closely resembles the pure mathematical definition

$$\text{fac}\ n = \begin{cases} 1 & (n = 0) \\ n * \text{fac}(n-1) & (n > 0) \end{cases}$$

The functional program does not describe precisely how to compute the factorial as efficient as possible on stock hardware as is common practice when coding in imperative languages like C [Kernighan78]:

```
fac(n)
int n;
{
    int f;

    f = 1;
    while (n>0) {
        f = f*n;
        n = n-1;
    }
    return(f);
}
```

A consequence of the *declarative* style of programming is that functional language compilers have to work harder than their imperative counterparts to generate efficient machine code, but in return it is much easier to apply program transformations, because "equals may always be replaced by equals" [Hudak89]. For example, in the following expression:

```
1/foo + foo
where
    foo = fac 481
```

the function application '`fac 481`' may be substituted for both occurrences of `foo` without changing the value of the expression. In general this is not the case in imperative languages where side effects can cause subsequent uses of `foo` to yield different values depending on the effects of the statements in between.

2.1 Reduction

A functional program can be considered a set of equations that are to be used to simplify a given expression (i.e. to solve a problem). This process of replacing expressions with equal, but simpler expressions is called *reduction*, and each simplification is called a reduction step. Reduction proceeds by repeatedly selecting a *red*ucible *ex*pression that matches the left-hand side of an equation and replacing the *redex* with the right-hand side of the equation. For example, the functional program:

```
square (1+2)
where
    square x = x * x
```

can be reduced as follows:

```
square (1+2)  ⟹  square 3  ⟹  3*3  ⟹  9
```

Reduction stops when no more simplifications can be applied to the expression that has reached its *normal* form and does not contain redexes anymore. The above sequence of reduction steps is not the only one possible to evaluate the expression '`square (1+2)`':

```
square (1+2)  ⟹  (1+2)*(1+2)  ⟹  3*(1+2)
              ⟹      3*3      ⟹     9
```

Although this second reduction sequence takes one more step than the first sequence it yields the same value ('`9`'). This is an important consequence of the Church-Rosser properties of the underlying lambda calculus: any order in which reduction rules are applied yields the same normal form, provided that the reduction sequence terminates. The *string* representation of the expressions caused the second reduction order to reduce the subexpression '`1+2`' twice. This "loss of sharing" could lead to an exponential amount of recomputation. Section 2.5 discusses the normal-order graph reduction strategy that has best termination properties, and represents expressions as graphs to minimise the number of reduction steps (i.e. to avoid the duplication of redexes).

2.2 Values and expressions

The pure lambda calculus deals with function definitions and function applications and nothing else. Even basic constants such as integers have to be represented as functions. Although this simplicity has appealing advantages in a theoretical framework, it makes programming awkward and error prone. Therefore practical functional languages include 'syntactic sugar' to denote constant values and simple expressions. For example, in Miranda characters are denoted with surrounding quotes (`'a'`, `'b'`, etc.), and integers can be combined into expressions with the usual set of infix and prefix operators.

In addition to basic data values, data constructors are provided to group logically related values together. These data constructors resemble records in Pascal or structures in C, and can be defined by the programmer as part of an *algebraic* datatype. The following example specifies a datatype that represents complex numbers and some operations:

```
complex ::= C num num

i = C 0.0 1.0
cadd (C ra ia) (C rb ib) = C (ra+rb) (ia+ib)
cmul (C ra ia) (C rb ib) = C (ra*rb-ia*ib) (ra*ib+ia*rb)
```

A complex number is specified as a data constructor named C that has two fields to hold the real and imaginary parts. To unravel their arguments, the function definitions of `cadd` and `cmul` use a notational convenience called *pattern-matching*. Patterns may be used to match arbitrarily nested and complex parameters, and constants can be used to select specific parameter values. For example the factorial function can be defined with pattern-matching as follows:

```
fac 0 = 1
fac n = n * fac (n-1),  if n>0
```

The boolean *guard* '`n>0`' is needed to limit the definition of the factorial to non-negative parameter values. If it would be omitted the expression '`fac (3-7)`' would not raise an exception, but cause an infinite computation with undefined behaviour.

The advantage of abstract datatypes is that the programmer may use the functions over the datatype without "knowing" the underlying implementation. Languages such as Miranda and Haskell include facilities to create abstract datatypes, whose implementation details can be explicitly hidden from the user. For example, the user of the datatype `complex` can perform complex computations by using the supplied operations without knowing of the exact data representation (i.e. data constructor `C`):

```
i_square = cmul i i
```

A powerful property of algebraic datatypes is that they can be used to specify recursive datatypes like lists and trees.

```
num_list ::= NIL | CONS num num_list

length NIL           = 0
length (CONS n lst) = 1 + length lst
```

The |-symbol denotes that an algebraic datatype is constructed out of several elements; a list of numbers is either the empty list, denoted by `NIL`, or is a number paired with the remainder of the list, which is another element of type `num_list`. The data constructors `NIL` and `CONS` are used by the function `length` to distinguish the two cases. Since it would be awkward to have to define a list for each basic type, algebraic datatypes may be parameterised:

```
list * ::= NIL | CONS * list

num_list  == list num
bool_list == list bool

length NIL           = 0
length (CONS x lst) = 1 + length lst

lst = CONS 1 (CONS 2 (CONS 3 NIL))
```

The `*`-symbol denotes a type variable that may be instantiated with an arbitrary type as in the declarations of `num_list` and `bool_list`. Now the function `length` operates on the *polymorphic* type `list` and can be used to compute the length of any kind of list since `length` does not access the individual list elements. In contrast, the function that sums all elements of a list can only be used for lists of numbers despite the syntactical resemblance with the `length` function:

```
sum NIL           = 0
sum (CONS n lst) = n + sum lst
```

2.2.1 Lists

Since lists are frequently used in typical functional programs, the Miranda language provides a built in notation for lists: `[]` denotes the empty list (cf. `NIL`), and the colon (`:`) serves as an infix pair constructor (cf. `CONS`). Thus the example list `lst` can be written as:

```
lst = 1 : 2 : 3 : []
```

An even more convenient notation is allowed by enumerating a list inside square brackets as a sequence of elements separated by commas:

```
lst  = [1,2,3]
lst' = [1..3]
```

The shorthand notation used to declare list `lst'` specifies an arithmetic sequence of values. This notation is frequently used in combination with list comprehensions since it is the prime means to specify repetition. *List comprehension*, also known as ZF expressions [Turner81], is the most powerful notation to specify lists and stems from the mathematical set notation. For example, the following expression computes all squares of the prime numbers in the range 1 to 100:

```
[n*n | n <- [1..100]; prime n]
```

It closely resembles the following mathematical set description:

$$\{n \times n \mid n \in \{1, \ldots, 100\}, \text{prime}(n)\}$$

The `prime` predicate can be defined with a list comprehension as well:

```
prime n = (divisors n = [1,n])
        where
           divisors n = [d | d<-[1..n]; n mod d = 0]
```

A prime number is only divisible by `1` and itself, but it seems like a waste to compute the complete list of divisors of `n` since as soon as the first non-trivial divisor `d` has been found, `n` is known to be not a prime number. Fortunately this is exactly what happens under lazy evaluation, which will be explained in Section 2.4, since the lists are compared element wise and the comparison stops when two unequal elements have been found.

2.3 Higher order functions

A fundamental concept of modern functional languages is that functions are first-class citizens: functions may be passed as arguments, returned as results, and stored in data structures just as ordinary data values like integers. A function that takes a function as an argument, or delivers one as a result, is referred to as a higher-order function. Traditional imperative languages like Pascal and C barely support higher-order functions: functions may be passed as parameters, but it is impossible to create 'new' functions at runtime by partially applying a function to some arguments. The ability to construct new functions out of existing ones provides great abstractive power to the user, and is commonly used in mathematics. The differential operator, for example, is a higher-order function that takes a function as argument and returns its derivative as the result.

$$\mathrm{D}f = f' \text{ where } f'(x) = \lim_{h\downarrow 0} \frac{f(x+h)-f(x)}{h}$$

This mathematical definition can be straightforwardly expressed in a Miranda program as follows:

```
diff f = f'
         where
             f' x = (f (x+h) - f x) / h
             h    = 0.0001
```

Note that this definition of `diff` crudely approximates the true derivative since it takes the limit by fixing `h` to a small constant; a better definition that employs a sequence of ever decreasing values will be presented in section 2.4. The important aspect of the example is that `diff` returns as its result a function, which is composed out of already existing functions (`f`, `+`, `-`, and `/`). The expression '`diff square`' approximates the function $f(x) = 2 \times x$ and can be used in more complex expressions. For example, the expression '`(diff square) 0`' yields an approximation to the derivative of `square` in point `0`: `0.0001`. Since '`diff square`' yields a function, it can be differentiated again: the expression '`((diff (diff square)) 0`' yields `2.0` as an approximation to the second derivative of `square` in point `0`.

An implicit way to create new functions is to partially apply a function to a number of arguments that is less than the arity of the function. This technique is known as *currying*, after the logician H.B. Curry, and can be used to specialise functions by fixing parameter values. For example, suppose that the binary function `deriv` is used to compute the derivative of a function in a given point.

```
deriv f x = (f (x+h) - f x) / h
            where
                h   = 0.0001
```

The expression '`deriv square 0`' yields the same value as the expression '`(diff square) 0`' since it can be shown that for any function `f` and point `x` the following equation holds: `deriv f x` $\equiv$ `(diff f) x`. The function `deriv` can be specialised to compute the derivatives of a specific function `f` by currying: '`deriv f`' is a valid expression, and is equivalent to '`diff f`', i.e. it represents the derivative of `f`. In essence the declaration of `deriv` may be thought of as syntactic sugar for `diff`.

Higher-order functions increase the expressive power of functional languages in comparison to languages that do not support functions as first class values. In addition, higher order functions can be used to gain modularity by glueing program parts together as argued by Hughes [Hughes89]. Higher-order functions can be defined to abstract out the common functional behaviour (i.e. the glue) of a program. For example, suppose that the functions `sum` and `prod` are defined to add and multiply the elements of a list as follows:

```
sum []     = 0
sum (x:xs) = x + sum xs

prod []     = 1
prod (x:xs) = x * prod xs
```

Both functions use a similar pattern to traverse the list and compute the result. This behaviour can be abstracted by introducing a higher-order function named `foldr` that captures the common parts and carries two parameters to account for the `0/1` and `+/*` variable elements:

```
foldr op val []     = val
foldr op val (x:xs) = op x (foldr op val xs)
```

Note that the infix operators have been replaced with a binary function parameter named `op`. The `sum` and `prod` definitions may now be changed into:

```
sum  = foldr (+) 0
prod = foldr (*) 1
```

The `(+)`-notation is Miranda syntax to convert an infix operator into an ordinary binary function. The `foldr` abstraction can be re-used for many other functions, for example, to test whether all elements in a list of booleans evaluate to true:

```
alltrue = foldr (&) True
```

For example, the expression '`alltrue [True,False,True,True]`' evaluates to `False`.

2.4 Lazy evaluation

Programming languages that provide non-strict semantics only evaluate those parts of the specified computation that are strictly needed to compute the final result of the program. This enables the programmer to define and use 'infinite' datastructures without causing the program to run forever as would be the case in traditional imperative languages. The class of non-strict languages includes many functional and logic programming languages. Such non-strict functional languages are usually called lazy functional languages, where the term lazy evaluation denotes the corresponding non-strict evaluation order.

An important advantage of lazy functional languages is that the programmer is freed from concerns about the evaluation order of expressions, and may separate data from control: it is possible to structure a program as a generator that constructs a large number of possible answers, and a selector that inspects only a few ones when determining the final answer. The increased power to modularise programs is probably the most important benefit of lazy evaluation.

Evaluating a functional program consists of repeatedly replacing a redex with the corresponding right-hand side of the matching equation. Frequently, however, the program contains multiple redexes and some strategy is needed to select the next redex to be reduced. Although it is guaranteed that all reduction strategies yield the same value upon termination, strategies differ in the number of reduction steps needed to compute the program result and termination behaviour. Two important reduction strategies are illustrated by the following example:

```
square (square (1+2))
```

Normal-order reduction always selects the leftmost outermost redex to be reduced:

```
square (square (1+2))  ⟹  square (1+2) * square (1+2)
```

Applicative-order reduction selects the leftmost innermost redex:

```
square (square (1+2))  ⟹  square (square 3)
```

The normal-order strategy is used for lazy evaluation, while applicative order corresponds with the traditional evaluation mechanism of (imperative) programming languages. The difference can be seen when expressions are evaluated that involve functions with non-strict arguments. A function is strict in some argument if the function result is undefined whenever it is called with an undefined expression for that argument.

```
inf = inf + 1
```

The expression '`inf`' is undefined since its evaluation causes an infinite sequence of reduction steps:

```
inf  ⟹  inf+1  ⟹  (inf+1)+1  ⟹  ((inf+1)+1)+1
     ⟹  ···
```

Nevertheless the following example that includes `inf` can be handled by normal-order reduction. The trick is to use `inf` in a non-strict context such as provided by `const`:

```
const 3 inf
where
    const c x = c
    inf = inf + 1
```

Normal-order reduction terminates after one reduction step since it immediately invokes the reduction rule for `const`:

```
const 3 inf  ⟹  3
```

Applicative-order reduction, on the other hand, first reduces the argument expressions of the `const` function and starts reducing `inf`:

```
const 3 inf  ⟹ const 3 (inf+1)
             ⟹ const 3 ((inf+1)+1)
             ⟹ ···
```

Since no expression is reduced unless the value is needed to compute the final answer, the normal-order reduction strategy of lazy functional languages relieves the programmer of explicitly stating the evaluation order of expressions: the programmer only has to describe what to compute. This allows for a programming style, known as circular programming [Bird84], where values may be declared before computed. Johnsson shows in [Johnsson87] how circular programming can be exploited, for example, in parsing with attribute grammars. Any attribute grammar can be straightforwardly translated into a set of functions, one function for each production rule, that state how each attribute has to be computed. These functions can be used for parsing immediately: lazy evaluation automatically evaluates the attributes in the right order.

2.4.1 Infinite datastructures

Lazy evaluation allows the programmer to specify infinite data structures (lists) and guarantees termination if only a finite number of elements is needed to compute the program's result. The following example returns the list with the first hundred prime numbers:

```
take 100 [x | x<-nats; prime x]
where
     nats = from 1

     from n = n : from (n+1)

     take n []     = []
     take n (x:xs) = [],                   if n=0
                   = x : take (n-1) xs,  otherwise
```

The list comprehension takes the infinite list of natural numbers (`nats`) and tests for each element whether it is prime or not. If not evaluated lazily, the program would diverge, but now the program properly stops after listing the first hundred prime numbers, since `take` deletes the computation of remaining primes when its counter drops to zero. This example demonstrates a typical coding style found in many lazy functional programs: a producer generates an infinite stream (list) of values, which is connected to a consumer selecting the appropriate items.

As a second example, we will derive a better definition for the `diff` function that differentiates continuous functions. The mathematical formula specifies that the derivative is the limit of a series of approximations, hence, the `diff` function can be declared as follows:

```
diff f = f′
   where
      f′ x = lim [(f(x+h) - f(x))/h | h <- approx 0]
```

The functions `lim` and `approx` still need to be defined. Note that the number of approximations is not known in advance since it depends on the required accuracy and the rate at which the function `f` converges. Therefore we will construct an infinite list of approximations, and have the `lim` function decide when an accurate value has been computed.

```
approx x = [x+dx | dx <- repeat (/2) 1]
repeat f x = x : repeat f (f x)
```

The expression ‘`repeat (/2) 1`’ constructs the decreasing sequence 1, $1/2$, $1/4$, ... by repeatedly halving the start value; the shorthand notation ‘`(/2)`’ denotes the function ‘`halve x = x/2`’. Taking the limit of the list of approximated derivatives can be accomplished by comparing the relative difference between two successive values:

```
lim (a:b:lst) = b,            if abs(a/b-1) <= eps
              = lim (b:lst),  otherwise
                where
                    eps = 1e-17
```

The lazy evaluation mechanism has allowed us to stay close to the mathematical description and separate the concerns of generating approximations and controlling the accuracy of the result, while these have to be merged into one thread of control for strict languages like C:

```
double deriv( f, x)
double f(), x;
{
    double x1, x2, h, fx, eps;

    eps = 1e-17;
    fx = f(x);     /* avoid recomputation */
    x2 = 0;        /* so test fails initially */
    h = 1.0/1024;
    do {
        x1 = x2;
        x2 = (f(x+h) - fx) / h;
        h = h/2;
    }
    while ( abs(x1-x2) > abs(eps*x1));
    return( x2);
}
```

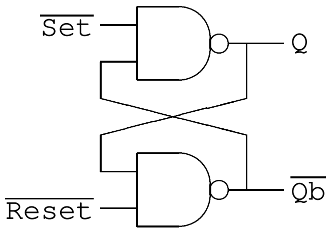
Note that the C program contains explicit code to avoid the recomputation of $f(x)$ since the C compiler cannot infer in general that function `f` has no side-effects. Because functional languages are side-effect free, the compiler automatically performs these kind of optimisations, for example, the expression '`approx 0`' is shared by all derivatives in a program.

Another advantage of the producer-consumer programming style is that both components (i.e. the `lim` and `approx` function) can easily be re-used in other programs or replaced by improved versions.

2.4.2 Stream programming

Lazy streams (lists) as supported by normal-order reduction provide powerful means to structure software: a large program can be composed of a number of processes interconnected by lazy streams. Such a stream process repeatedly consumes some elements from its inputs and produces a new output value. The processes may be glued together by an arbitrarily complex network of streams, and lazy evaluation takes care of selecting the appropriate process to produce a new stream value. The producer-consumer `diff` function is a simple example that consists of two processes connected by a single stream. An interesting example of a process network is the description of an elementary flip-flop circuit as shown below†:

†Reproduced from [Muller93] with permission of the author.



$\overline{\texttt{Set}}$	$\overline{\texttt{Reset}}$	$\texttt{Q}$	$\overline{\texttt{Qb}}$
1	1	$\texttt{Q}$	$\overline{\texttt{Qb}}$
0	1	1	0
1	0	0	1
0	0	1	1

Note that the flip-flop circuit contains a cyclic interconnection structure. It is, however, not necessary to worry about the cycles since lazy evaluation will do the trick. Therefore the diagram can easily be formulated as a functional program according to the *Kahn principle* [Kahn74]: First, label every stream in the network with a unique identifier. Then write down an equation for each stream, defining its value in terms of processes (functions) and other streams.

```
flip_flop set reset = q
                      where
                          q  = nand set qb
                          qb = nand reset q
```

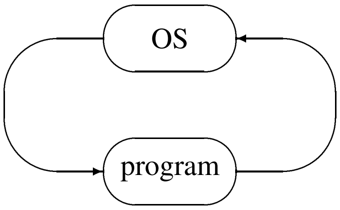
A `nand` process takes two streams of signals as input and combines them into one output stream. Signals traveling through a nand gate incur some delay, which is made explicit in the following function definition of `nand` that starts by producing two undetermined values (‘`X`’).

```
signal ::= X | H | L

nand as bs = X : X : (nand_op as bs)

nand_op (H:xs) (H:ys) = L : nand_op xs ys
nand_op (L:xs) (y:ys) = H : nand_op xs ys
nand_op (x:xs) (L:ys) = H : nand_op xs ys
nand_op (x:xs) (y:ys) = X : nand_op xs ys
```

This basic flip-flop can be combined with other gates and circuits to form arbitrary complex digital circuits. Another important application area of stream programming is in writing interactive software in pure functional languages. Early functional languages like Lisp include primitives to perform I/O like read and write system calls, but this violates the fundamental referential transparency principle and makes it difficult to reason about such opaque programs. Lazy streams can be used as follows to preserve the referential transparency when handling I/O:



Whenever the program needs to perform an I/O operation it “sends” a request message to the operating system (OS) on its output stream and “waits” for the response (e.g., the contents of a file) on its input stream. This set up makes the functional program referential transparent since given a stream of responses (i.e. input) the program always produces the same list of requests (i.e. output) no matter when the program is executed since all state handling is performed outside the program by the OS. Although the complete system (program + OS) is not referentially transparent, the program benefits from all the advantages like ease of reasoning and optimising program transformations.

Stream processing is not only an important concept for sequential programming, but it can be exploited as a parallel programming paradigm as well (see Chapter 3).

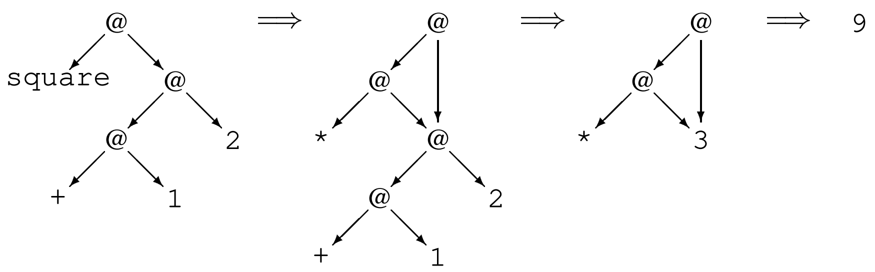
2.5 Graph reduction

The normal-order reduction strategy of lazy functional languages has optimal terminating properties: if a normal form exists then normal-order reduction will derive it. Unfortunately, naive implementation of normal-order reduction is inefficient since often redexes become duplicated. For example, the reduction of the expression ‘`square (1+2)`’ takes four steps:

```
square (1+2)  ⟹  (1+2)*(1+2)  ⟹  3*(1+2)
              ⟹      3*3       ⟹     9
```

The leftmost outermost selection procedure of normal-order reduction forces the subexpression ‘`1+2`’ to be reduced twice. The “loss of sharing” is caused by the *string* representation of the expressions, and can lead to an exponential amount of recomputation. Nevertheless several (parallel) functional language implementations have been designed with normal-order string reduction [Magó79, Kluge83].

To overcome the copying inefficiency of normal order string reduction, Wadsworth [Wadsworth71] proposed to represent expressions as graphs such that pointers to arbitrary redexes can be copied freely without duplicating work; whenever a redex is reduced, the result is shared by all pointers to the redex. This key idea of *graph reduction* makes the implementation of normal order reduction, or lazy evaluation, a practical technique. The reduction of the `square` example now uses the optimal number of reduction steps:

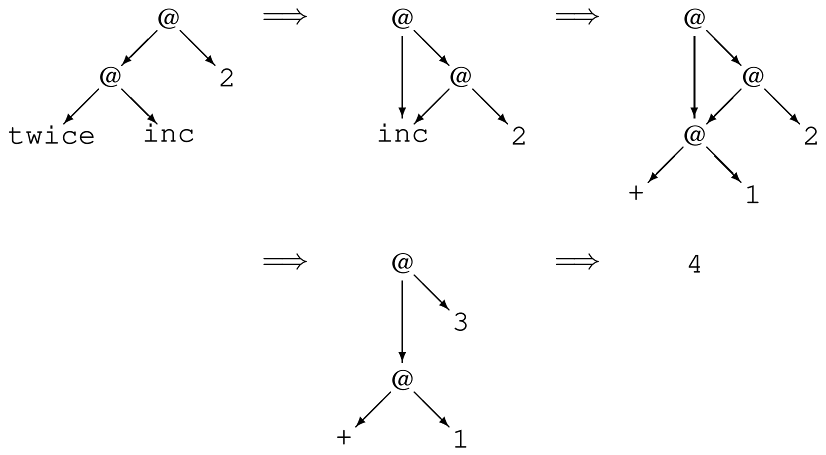


The @-symbol denotes function application; a unary function is applied to some argument. An expression that consists of a function applied to multiple arguments is handled by *currying*: a chain of function application nodes applies the function to all the arguments, one by one. For example, the expression '`+ 1 2`' is interpreted as short hand for '`((+ 1) 2)`', hence two @-nodes are required, where the '`(+ 1)`' expression denotes the function that increments its argument by one. This curried graph representation of expressions incorporates higher order functions without any difficulty since now functions can be passed around as ordinary pointers, and can be instantiated by adding application nodes.

In the following example the higher order function `twice` takes two arguments. The first argument is a function (`inc`) that is applied twice to the second argument (`2`), which may be another function.

```
twice inc 2
where
    twice f x = f (f x)
    inc = (+) 1
```

Note that the increment function `inc` is specified as a curried application of the built in addition operator `(+)`. The graph reduction of the term '`twice inc 2`' proceeds as follows:



A lazy graph reducer repeatedly performs the following steps: (1) find the leftmost outermost redex, (2) reduce the redex by instantiating the function body (i.e. build a graph), (3) update the *root* of the redex with the constructed graph.

Finding the next redex in step (1) starts by going down the left branch of each application node from the root until a function name is encountered. While *unwinding* the application *spine* the pointers to the application nodes are saved on a stack for subsequent use when rewriting the redex. After the unwind, the reducer checks whether the redex is an application of a function that evaluates its argument(s), like +, and tests if these so called *strict* arguments(s) have already been evaluated. If necessary the graph reducer recursively invokes itself to evaluate strict arguments before calling the function to rewrite the redex. At runtime the stack is used as with imperative languages, except that stackframes are not created at once, but instead are incrementally constructed when unwinding the spine.

The rewrite step (2) is important for the overall performance, and depends on the efficiency of instantiating a function body with the actual parameters. In [Turner79a] Turner published an implementation method based on combinatorial logics that was far more efficient at building graphs than the customary environment-based implementations derived from the SECD machine [Landin64]. A functional program is compiled into a small fixed set of elementary functions that only combine arguments without referring to other (global) identifiers. These simple functions are called *SKI combinators*, and the corresponding rewrite rules are incorporated as instructions of an abstract reduction machine. The next major efficiency improvement was to compile the program into application specific combinators, which are called *super combinator* [Hughes82]. This idea has been successfully implemented in the G-machine [Johnsson84, Augustsson84]. A program is first transformed through *lambda lifting* into a set of super combinators, which are then compiled into native assembly code for efficiency.

The update of the root application node of the redex in step (3) is needed to maintain the sharing of delayed computations. As a side-effect the references to the remaining graph nodes of the original redex are discarded, but the nodes cannot be reclaimed straight away since they might be referenced from other parts of the global computation graph. A garbage collector is needed to properly handle shared nodes when reclaiming garbage nodes in the heap. The presence of cycles in the computation graph complicates the garbage reclamation process [Cohen81].

To increase the performance of the basic graph reduction mechanism, the functional language compilers use numerous optimisations to avoid the construction and interpretation of graphs. For example, if the result of a single rewrite is an application spine then the graph reducer will immediately unwind the spine. Hence, the construction of the spine in the graph can be avoided altogether by pushing the arguments on the stack, and calling the function at the bottom of the spine directly.

For large applications lazy functional language implementations use much more (heap) memory than their imperative counterparts despite strictness analysis and other high-level compiler optimisations. At the low implementation level, space requirements can be cut down: tags can be encoded in a few bits in the pointer *to* the object instead of *in* the object itself (Chapter 5), and often chains of application nodes can be encoded in one vector apply node. These variable length vectors, however, complicate the allocation and reclamation of nodes in the heap. Reference counting or mark&scan garbage collectors have difficulty accommodating variable length vectors, so compacting garbage collectors that move live data into one contiguous block are used in general. To efficiently support garbage collection, several abstract graph reduction machines contain multiple stacks to separate heap pointers from other stack items like return addresses and basic data values (integers, floating point numbers, etc.). Multiple stacks are more difficult to manage, and the alternative is either to tag all values or to record the pointer positions in each stack frame.

A comprehensive description of the basic graph reduction principles and optimised implementation techniques can be found in [Peyton Jones87b].

2.5.1 Strictness analysis

Strictness analysis is an important optimisation technique that determines for each function which parameter values are needed to compute the result. As a consequence the *strict* arguments of a function may be evaluated safely before calling the function without violating the lazy evaluation semantics. Thus strictness analysis allows the compiler to use efficient call-by-value semantics for certain parameters instead of call-by-need semantics that forces the construction of graphs. This dramatically increases performance of lazy functional languages, for example, [Hartel91b] reports up to 92% reduction in claimed heap nodes when switching strictness analysis on.

A function is strict if the result cannot be computed when its argument value is undefined. Formally

$$\text{A function } \texttt{f} \text{ is strict} \quad \text{iff} \quad \texttt{f}\ \bot = \bot$$

The special $\bot$-symbol (called "bottom") denotes a non-terminating computation like the function `inf` defined as `inf = inf+1`. The job of the compiler is to determine for each function whether the above condition holds or not. Numerous (formal) strictness analysis methods have been devised [Abramsky87], but in essence these program analysis techniques may be thought of as propagating information through a syntax tree. For example, consider the strictness analysis of the following function:

```
divide x y = NaN,   if y=0    || return exception
           = x/y,   otherwise
```

The corresponding syntax tree is shown in Figure 2.1. The strictness analysis of `divide` can be performed by propagating information of the form "needs

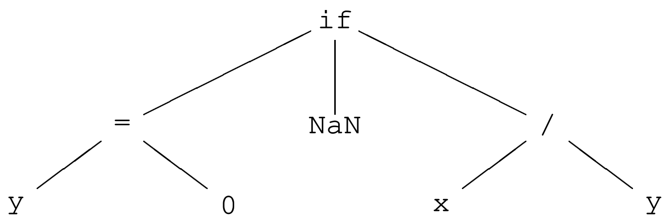


Figure 2.1: Syntax tree of function `divide`.

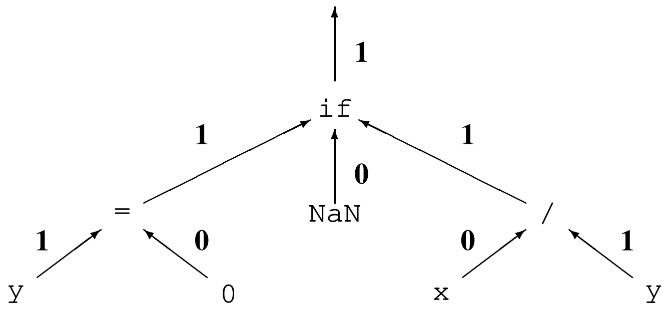


Figure 2.2: Strictness analysis of argument `y`.

`y`" and "may not need `y`" up through the tree. The result is shown in Figure 2.2, where **1** denotes "needs ..." and **0** denotes "may not need ...". The analysis shows that `f` is strict in `y` since the equality operator is strict in both arguments and the conditional expression of the if-statement is always executed. However, `f` is not strict in `x` since `x` is only used if the test fails in the conditional, see Figure 2.3.

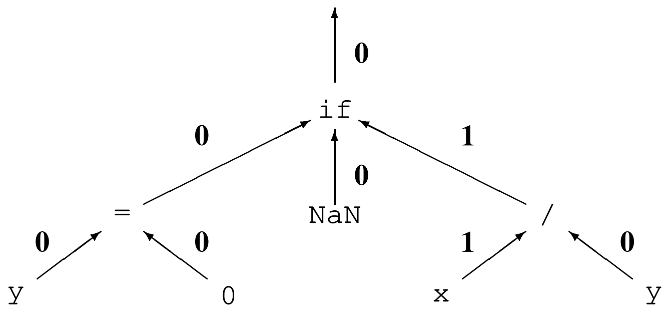


Figure 2.3: Strictness analysis of argument `x`.

The exact rules of how information propagates through the syntax tree are dependent on the operational behaviour of the basic operators and language constructs. Handling of non-recursive function calls is straightforward: first analyse the called function, then propagate information on strict arguments positions in the tree.

Recursive functions, which are frequently used, severely complicate strictness analysis since information is required that is being computed. The solution is to compute a number of successive approximations incorporating more refined information at each step. At first we assume that no information about the function itself is known (all arguments are non-strict) and propagate this information through the syntax tree to find a subset of the strict arguments. This information is used in a second traversal to find more strict arguments. The process is repeated until all strict arguments have been detected. Correctly determining the limit (or fixed point) of the successive approximations is a difficult and time consuming problem [Peyton Jones87b, Hughes90].

Although the above outlined strictness analysis methods determine which arguments are needed, the compiler must not completely reduce expressions at such strict argument positions since in case of datastructures it is not specified which components are needed, if any component is needed at all. For example, to test whether a list is empty, it satisfies to check that the list contains at least one element or not, but the value of the first element is not requested and must not be computed to preserve lazy semantics. Therefore the compiler may evaluate expressions to *head normal form* on strict arguments positions, but not to *normal form*. An expression is in head normal form if the corresponding root node in the graph is a constructor node, i.e. anything but an application node (redex). An expression is in normal form when the corresponding graph does not contain any redexes.

More powerful analysers extend their domain to account for strictness inside datastructures as well. The evaluation transformer model of strictness analysis [Burn91], for example, takes the needed structure of list-type expressions into account as well. Some functions like `length` and `append` require their (first) argument to be in spine normal form, that is the complete structure of the list is needed to compute their result. Others like `sum` even need the values of the individual elements, so whenever calling `sum` the argument expression may be evaluated to normal form in advance. Although this provides the compiler with more opportunities to pass the parameters by value instead as pointers to unevaluated expressions in the graph, the analysis time increases dramatically because the algorithmic complexity is exponentially proportional to the number of strictness properties (i.e. domain elements).

2.6 Summary

The short tutorial on functional programming and its implementation presented in this chapter provides the essential background information needed to comprehend the difficulties of functional language implementation on parallel machines, which is the topic of this book. The next chapter discusses the fundamental issues raised when extending graph reduction to parallel machines, and surveys a number of actual parallel implementations. The remainder of the book, chapters 4 to 6, describes the WYBERT approach to parallel execution of functional programs on shared-memory multiprocessors.

Chapter 3

Parallel implementations of lazy functional languages

Since functional languages are referential transparent, they are good candidates for programming parallel machines. The lack of side effects guarantees that any (parallel) computation order yields the same result (assuming termination). In principle the compiler can extract the parallelism from the program, and schedule it for execution on a given parallel machine. The accompanying run-time support system dynamically handles the resource allocation of individual computation grains: it allocates memory (garbage collection) and processing power (load balancing), and performs inter-processor communication. Thus, functional languages offer the prospect of releaving the programmer from the difficult task of parallelising a program. The compiler and RTS do the job for the programmer automatically.

The automatic extraction of parallelism in functional programs is based on properties of primitive operators and strictness analysis. For example, consider the expression $E1 + E2$; the strict semantics of $+$ allows for parallel execution of the expressions $E1$ and $E2$. Unfortunately, the implicit parallelism results in fine grain computations, which are difficult to execute efficiently on todays parallel hardware; even on shared memory multiprocessors, which offer low latency communications, the synchronisation of the huge number of fine grain computations results in a performance bottleneck. Some approaches try to enlarge the grainsize of the basic computations by automatic complexity analysis of expressions, see for example [Goldberg88c]. However, the usage of higher order functions and lazy evaluation, which are two key features for modular programming [Hughes89], severely limits the scope and impact of these compiler optimisations.

To overcome the problems associated with fine grain parallelism, a number of hardware architectures especially designed for executing parallel functional programs have been proposed [Magó79, Darlington81, Watson88, Waite91]. These designs typically support fast context switches to overcome communication delays when fetching "remote" arguments, and include packet switched communication protocols to transport the basic data unit (i.e. a graph node) as efficient as possible. Many of these designs have been inspired by developments in data-flow machines, but they never caught on since by the time they had been constructed –if constructed at all– the conventional-processor based implementations could use more advanced VLSI technology and clearly outperformed the special designs.

Instead of pursuing the "Holy Grail" of compiler derived implicit parallelism, recent research in parallel implementation of functional languages has taken a pragmatic approach. The programmer is required to explicitly annotate expressions that are worth to be evaluated in parallel. Then the runtime support system takes over and automatically schedules the resulting parallel tasks for execution and manages the machine resources (memory, processors, and communication). Thus in contrast to many imperative parallel programming languages, the programmer is only responsible for parallelising the algorithm, and does not have to handle low level issues like task placement, storage allocation, communication, and access control of shared data.

An important goal of many research programs is to show that the highly abstract parallel functional programming model can be implemented efficiently on stock hardware (in particular MIMD machines). The best results have been obtained for *strict* functional languages, which do not support lazy evaluation, like various LISP derivations: Multilisp [Halstead Jr84], QLisp [Gabriel84], and Mul-T [Kranz89]. Most notably are the results of the SISAL implementation that runs on a Alliant vector processor and outperforms the parallelising FORTRAN compiler on several large scientific applications [Cann92]. The efficient parallel implementation of *lazy* functional languages, however, is more difficult. The first prototype implementations on real parallel machines have been constructed, but few performance results for significant applications have been published.

This chapter looks at the parallel implementation issues of lazy functional programming languages; it focuses on the runtime support system since that is responsible for managing resources and parallelism in the machine. Section 3.1 discusses fundamental issues raised when extending sequential graph reduction to parallel machines: global (virtual) address-space support, generation and control of parallelism, storage management, and scheduling. Note that many of these problems also show up in parallel implementations of programming languages in general, but the discussion is limited to the scope of parallel graph reduction. Next, a number of recent designs and prototype implementations of parallel lazy functional languages is reviewed regarding these general issues; early parallel lazy functional language implementations have been described in [Treleaven82, Kennaway83, Vegdahl84]. Finally, a comparison between the surveyed machines is made in Section 3.3.

3.1 Parallel graph reduction

Graph reduction, as briefly discussed in the previous chapter, has been widely accepted as an efficient implementation method for lazy functional languages [Peyton Jones87b]. It is suitable for execution on parallel systems since the Church-Rosser property of the underlying lambda calculus guarantees that any reduction order yields the same result upon termination. In particular, several redexes may be rewritten concurrently, and the global (serialised) order in which the reductions are actually performed has no effect on the program's final result. This property allows a collection of graph reducers to rewrite redexes in parallel.

To the runtime support system (RTS), each graph reducer is a process (thread) that consists of a code segment, one or more stacks, and a large heap memory. The heap is shared by all graph reducers and has to be garbage collected occasionally by the RTS when the graph reducers run out of shared heap space. The RTS also has to schedule the graph reducers for execution on the parallel machine, and allocate memory for each reducer. The generation of parallelism and the cooperation between the multiple reducers is usually of no concern to the RTS because this is dealt with inside the graph reducers; all task synchronisation/communication is implicitly regulated through the program graph.

The following discussion of fundamental issues in parallel graph reduction systems does not cover I/O since the systems are targeted as compute engines connected to some host that handles the user interface: an expression is down loaded from the host, execution starts, and finally the result is transported back, hence, any standard communication protocol suffices.

3.1.1 Generating parallelism

To efficiently exploit the Church-Rosser property of graph reduction, care has to be taken to select the appropriate redexes for parallel execution since otherwise the graph reducers will rewrite redexes that are not needed to compute the program's final result. In the worst case, the superfluous execution of an infinite sequence of reductions results in a situation where all graph reducers are busy and fail to terminate properly. To avoid any waste of computing resources only expressions that are certainly *needed* may be evaluated in parallel (conservative parallelism); speculative parallelism [Burton85] is not considered since it is too difficult to manage. The selection of needed redexes for parallel execution is performed either by the compiler automatically, or by the programmer explicitly through annotations.

Compiler derived parallelism

Whenever a function is applied to a strict argument (see Section 2.5.1), the argument may safely be evaluated in parallel with the execution of the function body since at some execution point the argument value is needed to compute the function's result. Thus a strictness analyser, which is employed by the (sequential) compiler to transform call-by-need into call-by-value, also can derive the information needed to support conservative parallelism.

Unfortunately the effectiveness of strictness analysis methods is hampered by the usage of higher-order functions and data structures, as explained in Chapter 2. As a consequence the number of strict arguments that can be derived in a reasonable amount of compilation time is limited. Worse, however, is that the resulting parallel computations are fine grained: a single addition, one function call, etc. Efforts to automatically increase the grainsize have not been successful as will be discussed in Section 3.1.5. Fine grain parallelism is difficult to implement efficiently since overheads like scheduling, data communication, and context switching should be as small as possible, which calls for special hardware support.

Annotated parallelism

To overcome the problems associated with fine-grain parallelism, the programmer often has to assist the compiler by inserting annotations in the program source to explicitly denote coarse-grain expressions suitable for parallel execution. Two classes of annotations can generally be distinguished: skeletons and fork-primitives. They differ in the flexibility provided to the user and the amount of knowledge required from the user.

Skeletons provide the user with a high-level abstraction of a particular parallel programming paradigm like the divide-and-conquer or replicated-worker model [Darlington91]. The user just has to structure his program to fit one of the supported skeletons, and call the corresponding runtime support function to obtain parallel execution. The rigid control structure of the skeletons allows for efficient implementation on various parallel systems.

The Caliban annotation language [Kelly89, Cox92] supports a rather flexible skeleton: the *process network*, which consists of an arbitrary number of concurrent processes connected by (lazy) streams. Although the network description may be parameterised, for example by the number of processors, the description has to be compile-time static. This assures that an efficient fully-static distribution of the computation on the machine can be accomplished.

Fork primitives provide an unconstrained low-level method to start (spark) the evaluation of an arbitrary expression by putting a marker on it. The programmer has to take care to only annotate needed expressions, otherwise the program may incur superfluous computations or even fail to terminate. The following example shows a parallel merge sort algorithm where the annotation '`{!}`' expresses that recursive sorts of the left and right halves may be computed in parallel:

```
sort []   = []
sort [x]  = [x]
sort list = merge {!}(sort L) {!}(sort R)
            where
               (L,R) = split list
```

At runtime, a call of `sort` with a list of more than one element results in the creation of two *tasks*, which are placed in the global task pool that is consulted by idle processors looking for work. After *sparking* the child tasks, the parent continues execution, and when it requires the value of a sparked expression while the child task is still computing, the parent calls the RTS to block itself. When the child completes the evaluation and has updated the graph with the result, it notifies the RTS that the parent has become executable again. When the parent task is resumed, it starts with fetching the wanted result value from the graph. To avoid a sequence of expensive block/resume operations when the parent accesses the results of several child tasks in a row, a counter can be used to resume the parent only when the last child has completed.

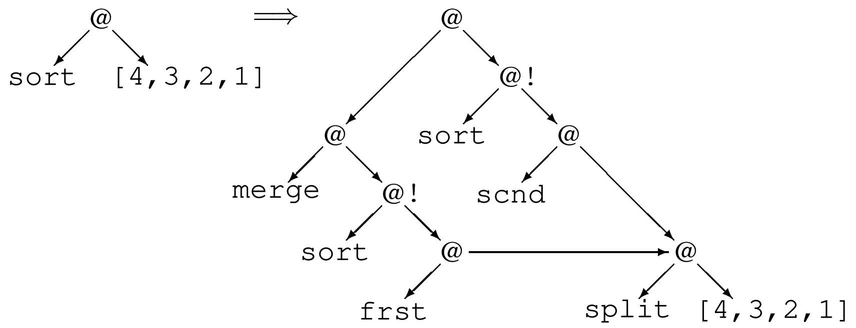
Examples of fork primitives are the futures in parallel LISP systems [Halstead Jr84], and the spark constructs in parallel graph reduction implementations [Peyton Jones87b]. These fork primitives give the user fine control over the parallel execution of a program, but require intimate knowledge about the underlying runtime support software and hardware architecture to obtain efficient execution. Some parallel implementations even force the programmer to resolve resource allocation issues like task scheduling and data communication. In the para-functional programming system [Hudak86], for example, the programmer has to specify on which processor a parallel task should be evaluated.

Most research in parallel implementations of functional programming languages is based on the low-level fork annotation since it provides coarse grain tasks as opposed to the fine grains of compiler derived parallelism. In addition fork primitives offer greater flexibility than the high-level skeletons; skeletons can be provided easily as library functions built out of fork primitives, but the other way round is much more difficult, if not impossible. Henceforth we will refer to the method of forking tasks and explicit waiting for results as the spark-and-wait model.

3.1.2 Global address-space support

Conceptually the spark-and-wait model of parallel graph reduction consists of a number of graph reducers that repeatedly rewrite redexes in parallel in different parts of the shared program graph. The activities between the reducers are coordinated in the RTS through a global task pool, which contains pointers to needed expressions in the graph, and the blocking/resumption mechanism discussed above.

The shared program graph in combination with a lazy evaluation mechanism complicates the implementation of functional languages on parallel machines, since tasks can easily share delayed computations that still have to be evaluated and updated. In case of the merge sort example, the redex '`sort [4,3,2,1]`' is rewritten as follows:



The `!`-symbols mark the two expressions that have been sparked for parallel execution. The `frst` and `scnd` primitives are inserted by the compiler to extract the `L` and `R` lists from the data structure (i.e. tuple) that will be returned by the `split` function. Note that the '`split [4,3,2,1]`' redex is shared by the two parallel tasks. Therefore special measures must be taken since otherwise the redex will be reduced twice, or worse, a task can read the partially updated root node of the redex and chaos results.

The presence of shared redexes in the global program graph requires special access protocols both on shared-memory and distributed-memory parallel machines, while the implementations on distributed memory machines are also faced with the problem of supporting a logically global address space.

Shared memory

The class of physically shared-memory parallel machines fits the spark-and-wait parallel graph reduction model very well: only the access of shared redexes has to be regulated. The usual solution is to slightly modify the sequential graph reducer. Each application node is extended with a lock (bit), which has to be acquired by a graph reducer before the fields of the node may be accessed. This solves the consistency problem. To avoid the duplication of work, each graph reducer marks the spine of application nodes it visits as "under reduction". Whenever a graph reducer requires the value of such a node, it blocks itself by linking its descriptor on a waiting list associated with that node. The update of a redex becomes slightly more complicated since the reducer has to check the waiting list and wakeup all suspended reducers so they can resume their computations.

The graph reducer encounters the overhead of setting and releasing a lock for every application node it processes, even though in practice only a small number of application nodes is shared. Measurements in [Hartel88b] report that typically 3-14% of the application nodes are shared in a Turner-combinator implementation [Turner79a]. It is possible to avoid some of the overhead by using a new kind of application node to denote non-shared application nodes that need not to be updated, hence, that need not to be locked. The classification of application nodes can either be done statically at compile time through update/sharing analysis [Peyton Jones92] or dynamically at run time by (one-bit) reference counting [Stoye84]. It is not clear, however, how successful these techniques are, and in particular whether the performance gain of the reduced locking and updating outweighs the overhead of reference counting.

Distributed memory

An important design issue is how to map the shared-data view of the spark-and-wait graph reduction model onto the message-passing based distributed memory machines. Two basic approaches can be taken: 1) a transparent layer of software on top of the bare hardware provides the graph reducers a single uniform addressable (virtual) address space, 2) the graph reducer is modified to explicitly deal with “remote pointers”, which have to be dereferenced by sending a message to the processor that holds the data.

Global virtual address space As discussed in Chapter 1, Shared Virtual Memory [Li89] has been developed to provide the user with a single global address space on distributed systems. It uses virtual memory techniques to intercept references to remote data and fault in the data by sending a message to the owning processor. Since the handling of inter processor communication is performed transparently by the operating system, the user program is presented the impression of operating on a shared memory machine.

Shared Virtual Memory has not been used directly as an implementation platform for functional languages because the granularity of a page is far too large: graph reducers operate on nodes that occupy a few bytes, not 4Kbyte, so pages bounce back and forth between processors when graph reducers repeatedly update different nodes that reside on the same page. This behaviour is known as false sharing. The distributed functional language implementations that do support a global virtual address space, like the Flagship machine [Watson88] and PACE [Waite91], operate on graph nodes instead of pages. These designs contain special hardware to map globally addressed nodes onto locally cached copies.

To avoid the duplication of work, shared redexes must reside only at one processor; shared redexes may not be copied on external requests, but have to be reduced first, or moved to the requesting processor instead. Furthermore, application nodes have to be locked and marked “under reduction” as in the case of true shared memory implementations.

Remote pointers Most distributed functional language implementations incorporate some kind of “remote pointer” to refer to data that resides on a remote machine. The graph reducer has to be able to recognise these remote pointers, and whenever it needs the value associated with a remote pointer it sends a message to the owning machine. When the reply comes back, the graph reducer creates a local copy of the value, and continues ordinary execution. To hide the communication delay, the graph reducer does not wait for the reply, but immediately continues with the evaluation of another task.

The servicing of external requests for local graph nodes is not trivial. Basic data values like integers and floating point numbers can be returned as is, but data constructors have to be checked for local pointers. These local pointers have to be converted to “remote pointers”. The request for a redex is even more complex since sending a copy back results in the duplication of work. Therefore, the redex is reduced to head normal form first, before the computed value is sent back to the requester. To guarantee consistency between graph reduction and message handling, either all redexes should be protected by locks, or the runtime support system can translate requests into high priority tasks that are enqueued for ordinary processing. In the latter case, only the access of the task pool has to be controlled by locks since effectively tasks are never de-scheduled partly during a graph update. Still all application nodes must be marked “under reduction” to avoid the duplication of work since an active task can trigger the evaluation of an application node that already was being reduced by a suspended task on the same processor.

The scheme above guarantees the sharing of computations, but the resulting data may be freely copied throughout the whole machine. In particular several copies of the same data can reside at one machine as caused by multiple dereferences of the same remote pointer: each request results in a new local copy of the remote data. This situation arises, for example, when processing a list of queries on a remote database where the root of the database is copied for each query. Precious bandwidth and memory are wasted. The solution is to use one level of indirection: upon receipt a remote pointer is stored in a local indirection node, whose address is passed to the graph reducer. This local pointer may be duplicated at will since when the graph reducer fetches the remote value, it overwrites the indirection node, and all references then share the same copy.

Communication between tasks is a weak point because data is transported a graph node at the time. Firstly, it is more efficient to send large messages that contain several graph nodes since the software overhead and communication delay are incurred only once, and less bandwidth is consumed. Secondly, sharing inside complex structured data is lost; for example, when fetching a remote cycle the local copy unfolds into an infinite list since each request returns a copy that refers to the original remote cycle. A potential improvement is to answer external request messages by transporting the complete graph rooted at the requested node. Wrapping up the graph into one message, however, is quite complicated as sharing should be maintained. Redexes must not be copied, but should be left at the owning processor. It is not clear whether this overhead can be tolerated, especially since the lazy evaluation mechanism is unlikely to force the evaluation of large data structures at once, so probably only small graphs are available for transport anyway. The PABC machine design [van Groningen92] incorporates such a copy policy, but no measurements are available yet.

To avoid the communication problems caused by shared redexes, the APERM machine [Vree89] normalises shared data before sparking tasks. In return for sacrificing some laziness, APERM can safely copy a complete subgraph of a task to a remote processor since it does not contain any (shared) redexes. Thus all 'remote' data is transported in one message, instead of a node at the time. This copy approach reduces communication overhead, and allows for local garbage collection since inter-processor graph references do not exist. To enhance performance it is proposed to construct a graph transporter unit in hardware that wraps a graph into a single message and sends it to a remote node; it is conceivable that with minor extensions this unit can be used for local garbage collection (copying) as well.

3.1.3 Storage management

Closely related with the global address-space support is the storage management system, which allocates storage for graph reducers in the parallel machine. A graph reducer uses two kinds of dynamic storage: one or more stacks and a single shared heap. The heap is used to allocate graph nodes and is shared between all graph reducers, while each stack is accessed by one graph reducer only. The storage management system partitions the available physical memory into heap and stack space, and controls the allocation and deallocation of storage in both spaces.

Stack management

The graph reducer uses at least one stack to hold the arguments of nested function calls, but often multiple stacks are used to ease garbage collection by separating heap pointers from other stack items like return addresses and basic data values (integers, floating point numbers, etc.). The depth of the stacks varies considerably during reduction, while the maximum depth is generally unknown at compile time. Since the number of sparked tasks is also unknown beforehand, the storage management system has to accommodate an arbitrary number of stacks, whose size changes dynamically.

A straightforward solution is to equip each task with small stacks initially, and enlarge a stack each time it overflows. This approach has been taken, for example, in the PABC machine, where stacks are allocated as ordinary heap objects. On stack overflow, a new larger stack is allocated in the heap, the contents of the old stack is copied to the fresh one, stack pointers are adjusted, and reduction continues. The old stack space is reclaimed automatically by the garbage collector, but in return the (live) stacks have to be copied on each compaction of the heap. The amount of space added on each stack reallocation, controls the balance between the number of reallocations and the amount of wasted memory. As reported in [Kesseler91], an increase of just 10% “seems to work quite well” (on one Transputer).

An alternative solution found in many parallel graph reduction machines is to replace the monolithic stack by a linked list of stack frames, which are allocated in the heap. The size of each stackframe can be determined statically at compile time as described in [Lester89b], so stack overflow inside a frame never occurs, hence, costly reallocations are not needed. Like with the previous approach, deallocated stack frames are automatically reclaimed by the garbage collector, which is more expensive than re-using space on a conventional stack. Another disadvantage is that the locality in references to heap allocated stack frames is less than with a conventional stack outside the heap, which gives a performance penalty in parallel systems equipped with caches and on systems with fast local memory like the Transputer. Unfortunately, no measurements are available to quantify the exact costs of allocating stack frames in the heap.

Heap management

The distributed management of the global heap is difficult since graph reducers consume large quantities of heap space to allocate graph nodes with a short average life-time, so a garbage collector has to be invoked to reclaim the storage that is no longer referenced by the program graph. Research in garbage collection has a long history, and many different algorithms have been designed [Cohen81]. Three important classes of garbage collection algorithms can be distinguished: reference counting, mark&scan, and copying collectors. The copying collectors are most suitable for modern style graph reducers since these algorithms facilitate the fast allocation of variable sized, but small, nodes by compacting the live data into one consecutive block; a node allocation just amounts to advancing the free space pointer by the size of the node. Another advantage of the copying collectors is that they only traverse live data, which usually accounts for just a small fraction of the total heap space. In addition, copying collectors smoothly handle cycles in the program graph in contrast to reference counting collectors. Cycles can be banned from the graph, but at the expense of efficiency (unrolling) and expressive power (no cyclic data structures): for example, the elegant *circular programming* style [Bird84] can not be supported. Efficient copying garbage collection algorithms for parallel graph reduction machines are quite different for shared-memory and distributed-memory implementations.

Shared memory Since shared memory multiprocessors consist of a relatively small number of processors in general, it is viable to synchronise all processors when a garbage collection is needed. Once the processors have stopped, an ordinary sequential copying collector can reclaim all garbage [Augustsson89b], but with a minor adaptation all processors can participate: before inspecting a (live) node, the node has to be locked to properly handle sharing. This straightforward method is, for example, used in Multilisp [Halstead Jr84] and GAML [Maranget91].

Instead of having all processors collect garbage in parallel, it is also possible to arrange for one copying collector to work in parallel with multiple graph reducers (mutators). The method of *concurrent* garbage collection as described in [Appel88] uses virtual memory hardware to synchronise the collector and the mutators. When the mutators run out of free space, they copy their root nodes to to-space. Then the collector marks all pages in to-space as inaccessible to the mutators, and starts to scan the nodes in to-space to find references to live nodes in from-space. Whenever the collector has scanned a page in to-space, it marks the page as accessible to the mutators. When the collector starts scanning, the mutators immediately continue with ordinary graph reduction. If a mutator tries to reference an object that resides on a not-yet-processed page, the hardware generates an access-violation trap. This triggers the collector to handle that page immediately, after which the mutator resumes execution.

Distributed memory The inherent global nature of copying garbage collectors makes them unsuitable for distributed systems for two reasons:

1. All processors have to synchronise before garbage collection can start since nodes will be moved during the compaction phase; global synchronisation is expensive on large machines.
2. A message has to be sent for each remote pointer to learn the new location of that object; this results in a burst of data transport, which severely stresses the communication network.

Therefore the early distributed functional language implementations/designs have resorted to reference counting algorithms.

Distributed reference-counting garbage collectors show good locality since the bulk of increment and decrement operations are performed on graph nodes that reside in the processor's local memory. Only the copying and deletion of remote pointers require inter-processor communication. A serious constraint for large distributed systems is that the communication network must preserve the message ordering since the interchange of an increment and decrement message can lead to the incorrect reclamation of a graph node. The weighted reference counting technique [Bevan87, Watson87a] tackles this problem, and decreases the number of messages as well, by maintaining a weight with each pointer. The summed weights of all pointers to an object equals the reference count in that object, always. When a pointer is duplicated, the original weight is split between the two resulting pointers without the need to increment the reference count, hence, no message has to be sent. When a pointer is discarded, the reference count has to be decremented by the weight, and a message has to be sent in case of a remote pointer. Since weighted reference counting only uses decrement messages, the communication network may deliver messages in any order.

Unfortunately, reference counting has several disadvantages: 1) reclamation of cyclic structures is cumbersome [Brownbridge85, Hughes83]; 2) variable sized nodes are poorly supported; 3) the performance is less than that of copying collectors [Hartel90]. To overcome these problems, Lester

[Lester89a], has devised a composite approach: local garbage is reclaimed through copying, while weighted reference counting is used to handle global garbage. When a processor runs out of free space, it starts a copying collector to reclaim the garbage nodes in its local heap. Two indirection tables are used to handle remote pointers. The input-indirection table contains a (local address,reference count) tuple for each node that is accessible from another processor. The collector processes all entries with a non-zero reference count, and updates the local addresses when compacting live data in the heap. The input table is an array located at a fixed address, so the compaction is transparent to the external references on remote processors. The output-indirection table contains an entry for each remote pointer to a node that resides on another processor: a global address and a weight. The output table is implemented as a linked list. This list is traversed after each compaction to find remote pointers that are no longer in use, and decrement messages are sent to the owning processors.

Lester's composite algorithm elegantly integrates the locality of weighted reference counting with the efficiency of copying collection, but it can not handle cycles that span multiple processors. Rudalics [Rudalics86] presents a similar composite approach, but it uses a copying algorithm at the global level instead. This algorithm deals with global cycles at the expense of complex global processor synchronisation.

3.1.4 Task scheduling

The second important task of the runtime support system, besides memory management, is the distribution of work in the parallel machine. Each sparked task has to be scheduled for execution on a specific processor, or conversely, each processor has to be assigned a task when it becomes idle. For statically structured applications, scheduling decisions can be determined at compile time, but usually tasks are scheduled dynamically by the runtime support system. The dynamic scheduling of tasks on a parallel machine, also known as load-balancing, is a general problem that has received wide attention.

From a theoretical point of view, and if communication delays are neglected, any list scheduling policy will do since the resulting parallel execution time never exceeds twice the optimal execution time [Graham69]. (A list scheduling policy, or work-conserving scheduling discipline, is one that never leaves a processor idle when there is a runnable task available somewhere.) Furthermore, for applications with a high average level of parallelism, these scheduling policies achieve good processor utilisation as well; in [Eager89] it is shown that under any work-conserving scheduling discipline the processor utilisation is at least $A/(n + A - 1)$, where n denotes the number of processors and A denotes the average parallelism of the application (i.e. the speed-up factor on a machine with an unbounded number of processors and no communication delay).

Despite their nice theoretical properties, in practice list scheduling policies do not qualify for controlling large parallel machines for two reasons. First, a list scheduler needs global knowledge to maintain the invariant that no processor may be idle if there is a runnable task available somewhere. Hence, the creation and termination of each task has to be announced globally, which leads to network congestion in distributed systems, and to memory contention in shared memory systems. Secondly, only large tasks should be selected for remote evaluation in order to overcome data transportation costs; it is better to let a processor run idle than to allocate a task whose computational requirements do not outweigh the communication costs. Both for shared-memory and distributed-memory implementations it is important to employ a scheduling policy that achieves good *spatial* locality in combination with a high degree of processor utilisation.

Shared memory

In small shared memory systems, schedulers that operate on a single shared task pool are conceivable, but such a simple solution will not scale when adding more processors. For example, in the Buckwheat implementation [Goldberg88b] measurements showed that the access contention for the global pool slightly degraded absolute performance when going from 7 to 8 processors running in parallel. Buckwheat successfully employed a two-level queue structure to reduce contention; clusters of processors share a primary queue that overflows into the secondary queue, which is accessible to all processors in the multiprocessor. This arrangement also enhances locality since processors first access the primary queue before resorting to the secondary queue, hence, most tasks are allocated inside a cluster. This is important for large shared memory multiprocessors that have non-uniform memory access times. For example, in a bus based multiprocessor whose processors are equipped with caches, it is advantageous to execute a child task on the same processor as its parent since part of the data already resides in the local cache, hence, cache misses are avoided in comparison to remote execution.

Preemptive scheduling of tasks, which bounds worst-case behaviour, is not employed in parallel graph reduction implementations on shared memory machines, although occasionally tasks have to be stopped in a consistent state to participate in a global garbage collect. Task migration, on the other hand, is used by most implementations; when a blocked task becomes runnable after the requested value has been evaluated, it can be resumed at any processor since its state (i.e. the stack) resides in shared memory, but for performance it is better to resume execution at the original processor since (part of) the context of the task still resides in the cache.

Distributed memory

In large distributed systems it is not possible to use a central scheduler that controls all task allocations since it would become a bottleneck. Therefore scheduling decisions have to be taken locally based on incomplete information of the global system state. One solution is to use programmer annotations to control the task placement decisions [Hudak86]. Most distributed functional language implementations, however, provide automatic scheduling.

A popular distributed scheduling algorithm is called *diffusion scheduling* [Goldberg88a]. Each processor maintains an estimate of its own workload and communicates this to its direct neighbours on regular intervals. If the processor detects that its own load is significantly higher than of some neighbour, it off-loads some local tasks to that neighbour to balance the system load. As a consequence, work 'diffuses' across the parallel machine from busy parts towards lightly loaded parts. Satisfactory results are obtained for moderately sized machines, even though the workload estimate is usually based on the number of runnable tasks, not the actual computational demands.

A strong disadvantage of diffusion scheduling for large scale systems is that work spreads slowly across the machine: one hop at the time. Especially at the initial stage, when most processors are idle, it is important to allocate (large) tasks far away from their origin. That, however, requires global knowledge about the system state, which can not be maintained accurately at all processors. Hierarchical schedulers combine complete local information with general global knowledge by placing a tree shaped control structure on the machine: Processors are grouped into clusters, clusters are grouped into domains, etc. Each scheduler in the hierarchy maintains some global information about its sub-domains, and is authorised to move tasks between sub-domains. In the HyperMachine, tasks are scheduled at different levels according to their work estimates as provided by programmer annotations [Hofman92a]; this assures that only large tasks, which generate lots of work, incur high transportation costs.

Like in shared-memory implementations, distributed schedulers do not preempt tasks once they have started execution to avoid consistency problems in the heap. Task migration is also not supported since wrapping up the task state is complicated, and migration is only beneficial if no freshly sparked tasks are available. The latter situation can be circumvented by generating a lot more tasks than the number of processors in the parallel machine. Sparking tasks is not for free, so it might be better to keep machines idle for a short time instead of generating too many tasks.

3.1.5 Controlling parallelism

Although generating many tasks eases the load-balancing of the parallel machine and leads to high processor utilisation, it is not for free. There is always some overhead associated with the sparking of a task, so tasks must have some controlled minimal size that outweighs the overhead. Furthermore, the number of tasks should be controlled too because uncontrolled breeding of tasks exhausts the machine resources: task pools overflow, heaps fill up, etc. Ideally, an application unfolds into one large task per processor, which can be evaluated independently in local memory, before the results are combined into the final solution. In practice, of course, this ideal is rarely accomplished, but effective strategies have been devised to control the grainsize and the number of tasks in parallel graph reduction systems.

Grain size

The minimal amount of work of individual tasks has to exceed a given value to overcome the overhead costs associated with sparking:

- The expression has to be constructed as a graph, which is more expensive than call-by-value evaluation.
- The task descriptor has to be placed in a task pool so it can be scheduled for execution.
- The task has to be transported in case of remote execution.
- A new graph reducer has to be started, i.e. a context switch occurs, when the task is scheduled for execution.
- The list of waiting reducers has to be signaled to resume execution when the task has been evaluated.

Efforts have been undertaken to have the compiler determine automatically through *complexity* analysis whether a task is big enough to justify the spark. Of course, heuristics have to be used since the problem is undecidable in general, for example, expressions can depend on input data. Goldberg [Goldberg88c] handles recursive function calls and the invocation of higher order functions by assigning an infinite cost to them. Unfortunately, this approximation assigns an infinite cost to most tasks encountered, so few sparks are avoided in practice. Compile-time complexity analysis seems incapable of generating coarse-grain tasks; it can merely enlarge the small grain size of automatically derived parallelism with a small amount.

Like with generating parallelism, the programmer has to do the job. Insight in the algorithmic complexity of the application allows the programmer to place spark annotations at large expressions only. For example, in the merge-sort program the list is recursively divided into two halves until the empty list results; each division creates two more parallel tasks. The sparking of these tasks is denoted with the '`{!}`'-annotation in the code below. It is rather straightforward to stop sparking tasks when the length of the list falls below some threshold:

```
sort []   = []
sort [x]  = [x]
sort list = if (#list > 37)
              merge {!}(sort L) {!}(sort R)
            else
              merge (sort L) (sort R)
            where
               (L,R) = split list
```

Some fine tuning based on performance measurements is needed to determine a suitable cut-off threshold. To avoid the overhead of repeatedly computing the complexity measure at each recursive invocation, two versions of the function are used: a parallel version that sparks tasks for complex calls, and jumps to the efficient sequential version without sparks/tests otherwise. A disadvantage of this optimisation is that once a task switches to sequential code it will never spark a task again, even when the load drops to zero in the future.

Number of tasks

To avoid exhaustion of machine resources, the number of tasks has to be controlled as well as the grain size. The cut-off strategy of the previous section can be refined to take the machine load into account to stop sparking when enough parallel tasks have been created. In Qlisp, for example, the programmer is provided with a set of primitives that return system parameters like the number of processors, and the queue depth (i.e. the number of runnable tasks on a processor) [Pehousek89]. A more sophisticated solution is to have the runtime support system decide whether or not to spark a task based on the programmer's complexity information that indicates the amount of work involved in evaluating that task. In case of merge sort the time needed to sort a list is in the order of $n \log n$ operations, where n is the length of the list. Hence, we can use the length of the list as a rough indication of the grainsize of a task:

```
sort []   = []
sort [x]  = [x]
sort list = merge {!sz}(sort L) {!sz}(sort R)
            where
               (L,R) = split list
               sz = #L
```

The `{!sz}` annotation provides the complexity measure to the runtime support system, which can combine this information with system parameters like the processor load to control dynamically the number of tasks by inhibiting sparks. The HyperM scheduler [Hofman92a] uses programmer complexity annotations to assist allocation decisions.

The GRIP machine [Peyton Jones89] takes a radically different approach to avoid flooding the system with tasks: it discards tasks when the task pool fills up. To guarantee that the work of the discarded tasks will eventually be done, the GRIP machine uses the evaluate-and-die model of parent-child synchronisation that differs slightly from the spark-and-wait model discussed in section 3.1.1: When a parent needs the value of a child task that has not started evaluation yet, it does not block itself as usual, but rather evaluates the expression itself. The child task becomes an orphan and can be discarded. As a consequence, tasks may be deleted from the spark pool without further notification since the parent task will always try to evaluate the associated expression itself later on. Likewise, access of the global task pool is not protected by a lock since loosing and/or duplicating a few tasks does no harm as can be seen from the preliminary performance measures in [Hammond91].

If the runtime support system has no means to control the sparking of tasks, as is the case in many parallel functional language implementations, the dynamic number of tasks in the machine can still be regulated by a scheduling heuristic. We discuss a heuristic that works particularly well for divide-and-conquer applications.

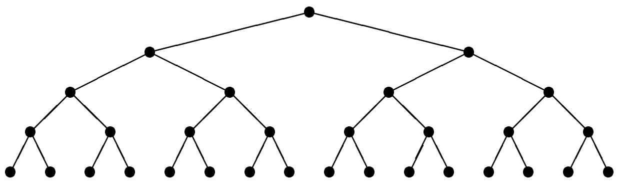


Figure 3.1: divide-and-conquer task structure.

Divide-and-conquer applications unfold into a tree shaped task control structure with worker tasks at the leafs. If the task selected for execution is the most recently sparked task, then the tree is explored in a depth-first strategy; conversely, if the least recently task is scheduled, the tree is traversed breadth-first. The depth-first exploration (LIFO) requires the least amount of resources since it minimises the number of waiting control tasks, while the breadth-first strategy (FIFO) maximises parallelism. The Manchester dataflow machine used this observation to *throttle* the parallelism by dynamically switching between breadth-first and depth-first schedulers for low and high work loads, respectively [Ruggiero87].

Many parallel graph reduction machines have incorporated the throttling idea. A straightforward implementation on shared memory systems uses a linked list for the task pool with a scheduler that employs a LIFO access policy. This simple strategy minimises resource usage since it traverses the task tree in a global depth-first like order, but it does not take any locality considerations into account and produces schedules with many synchronisation points. A global LIFO scheduling on a four node machine of the divide-and-conquer application in Figure 3.1 is shown in Figure 3.2 where the nodes are labeled with the processor number. Note that all 15 parents have a remote child.

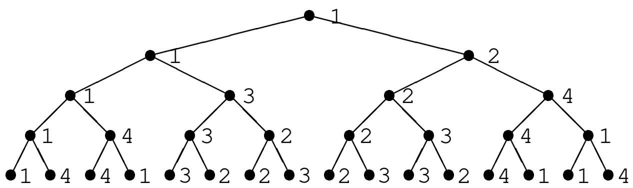


Figure 3.2: Processor assignment by global LIFO scheduler on a four node machine.

An improved method, which can also be used in distributed implementations, maintains a local task pool at each processor. Processors schedule their tasks in a LIFO order, but when running idle a processor steals a task from another processor's task pool in FIFO order. This strategy enhances efficiency since, after a short initialisation phase, processors evaluate different sub trees in depth first order, and minimal communication and synchronisation between processors is needed because only large tasks will be stolen, see Figure 3.3 where only three tasks are exported.

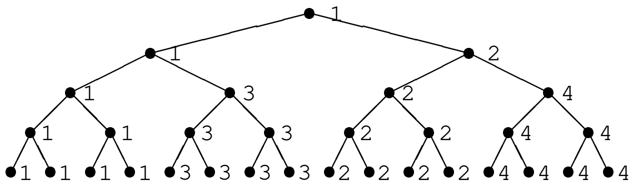


Figure 3.3: Processor assignment by local-LIFO/steal-FIFO scheduler on a four node machine.

The "Lazy Task Creation" method [Mohr91] even takes this idea further and avoids some of the sparking overhead. During ordinary execution, tasks are not sparked, but evaluated eagerly instead (LIFO); each task invocation leaves a frame on the call stack, where execution continues once the task has been evaluated. When a processor becomes idle, it *lazily* creates some task by stealing the oldest continuation frame (FIFO) from a busy processor, which involves patching the original frame to store the computed value in a place holder. The idle processor "continues" the original execution, and eventually synchronises on the place holder when it needs the task value. Stealing a continuation is more expensive than allocating a task, but these costs are only incurred when work is actually requested, while tasks are created in large numbers to avoid scheduling anomalies in advance. Preliminary performance results for Mul-T implementations show that "lazy task creation" performs better than ordinary spark-and-wait [Mohr91].

3.2 Survey

In this section we discuss a number of recent parallel graph reduction machines; early parallel graph reduction machines have been described in [Treleaven82, Kennaway83, Vegdahl84]. We review the most important design decisions taken in each machine regarding the issues raised in the previous section:

generating parallelism, global address-space support, storage management, task scheduling, and controlling parallelism. The selected parallel machines have been implemented on a wide range of hardware configurations. First, three shared memory machines are presented: $<\nu,G>$, AMPGR, and GAML. Then, four distributed machines are discussed: Flagship, PAM, HDG, and PABC. The latter three are all Transputer based implementations. Finally, we present two hybrid machines built out of shared memory clusters: GRIP and HyperM. A comparison of these nine parallel machines is given in Section 3.3.

3.2.1 $<\nu,G>$

The $<\nu,G>$-machine [Augustsson89b] is a parallel graph reducer for shared memory multiprocessors, and has been implemented on a 16-node Sequent Symmetry. It is an extension of the sequential G-machine implementation for Lazy ML [Augustsson84, Johnsson84], and supports the spark-and-wait model of parallel graph reduction; the programmer has to insert spark annotations in the LML source to denote opportunities for parallel execution.

The $<\nu,G>$-machine is based on one global address space as directly supported in hardware by the Sequent, which does include caches but no local memory per processor. The Sequent's ability to use any memory cell as a lock has been exploited to enforce mutual exclusive access on vector apply nodes, which are called FRAME nodes. These FRAME nodes are used in the $<\nu,G>$-machine to implement the calling stack as a linked list of frames in the heap instead of one monolithic stack as in the sequential G-machine. Each FRAME holds a delayed computation: a function code pointer, the right number of arguments, and some free space for temporary variables that are needed when the function is invoked. The FRAME sizes can be computed at compile time except when higher order functions are involved, in which case FRAMEs need to be extended occasionally at run time through allocating a new frame and overwriting the old one with an indirection node.

The original $<\nu,G>$-machine contains a straightforward heap management policy. Each graph reducer allocates large chunks (i.e. pages) of memory until heap space exhausts. Then a sequential garbage collector (GC) is invoked to reclaim heap space, after which reduction continues. Even with this simplistic scheme it is noted that "nevertheless GC seldom exceeds 30% of the total time." Recently an enhanced version of the Appel-Ellis-Li concurrent collector [Appel88] has been incorporated, which is capable of deleting speculative tasks as well [Röjemo92].

Each processor maintains a local run queue of executable processes. If the run queue is empty, a task is taken from the global task pool, and a process is created for it. The global task pool is not guarded with a lock to reduce overhead, so tasks may be lost occasionally. No work is lost since the evaluate-and-die model is employed; the parent task will check and evaluate the child itself in case of a loss. As a consequence, on average 10% of a processor's time is spent on task management overhead.

No provisions to control the grainsize or number of tasks are included in the $<\nu,G>$-machine; the programmer is solely responsible for controlling parallelism. The reported performance results for small annotated programs like nfib and 8-queens show speed-up over the pure sequential G-machine implementation: speedups range between 5 and 11 on the 16 processor Sequent Symmetry. The level-off in the speedup curves is attributed to bus contention, which might be caused by the poor locality of stacks allocated as linked lists in the heap.

3.2.2 AMPGR

The Abstract Machine for Parallel Graph Reduction [George89] is another shared memory implementation based on the sequential G-machine. A prototype implementation has been constructed for the BBN Butterfly multiprocessor, which consists of a number (15) of processing elements (MC68020 + 4Mbyte memory) interconnected through a delta network. Each processor can access transparently all memory in the whole machine, but local references are at least five times faster than references to remote memory. Therefore locality considerations have been taken into account in the AMPGR design.

The AMPGR machine, unlike the $<\nu,G>$-machine, is driven by compiler derived parallelism based on strictness analysis. The compiler inserts explicit spark instructions in the G-code whenever it can derive that a particular expression is needed; corresponding wait instructions are generated prior to the actual usage of the expression. To overcome the synchronisation overheads associated with large numbers of fine-grain tasks, many sparks are evaluated in-line without placing a description in the global task pool according to the following observations:

1. No evaluation is required for an expression in head normal form; no task is generated.
2. It is profitable to execute one child at home; the first spark inside a function is always neglected, so the reducer will evaluate the expression

itself after the other tasks have been sparked.

3. The total number of tasks should be limited; when the fixed sized task pool overflows, tasks are evaluated immediately.

Once the machine has reached a stable state, i.e. the task pool is full, the computation switches automatically to sequential execution; the only overhead incurred is the checking of in-line conditions.

To avoid contention, a two-level scheduling strategy is employed with local pools spilling over into a centralised pool. It has been found that the local pools should be small, 2 to 4 tasks, to keep all processors busy. The heap allocation is also distributed: a graph reducer first tries to allocate a chunk of heap space in the local memory, before looking for available space on remote processing elements. The AMPGR, however, does not include a garbage collector! For efficiency, each processor has been allocated a single stack in local memory; whenever a task has to be suspended, its context on the stack is saved in the heap, so the stack can be used by another task.

A running implementation on the BBN Butterfly has been constructed that shows good speed-ups, which is credited to the not so "blazingly fast" sequential implementation.

3.2.3 GAML

The GAML machine [Maranget91] is the third shared memory implementation that is based on the sequential G-machine. It is closely related to the $<\nu,G>$-machine since it also relies on programmer annotated parallelism, uses comparable compiler technology, employs the same scheduling strategy, and runs on similar hardware as well: the Sequent Balance (8 processors). Therefore we restrict ourselves to discussing the main differences between GAML and its relative the $<\nu,G>$-machine.

To reduce the overhead associated with synchronising multiple reducers in a shared address space, the GAML design includes a new node type to denote possibly shared nodes. Only these nodes have to be locked and marked ”under reduction” by the graph reducer, while ordinary application nodes can be accessed as efficient as in the sequential G-machine.

In contrast to the $<\nu,G>$-machine, GAML does not use a linked list of call frames, but incrementally allocates large chunks of stack space. When the stack shrinks the linked chunks are explicitly deallocated. To keep the number of stacks manageable, the sequential G-machine stacks are merged into one, hence, there is only one real stack per task.

The heap is managed through a parallel copying garbage collector. If a processor runs out of heap space, it enters a wait loop until all processors have stopped with graph reduction. Then the heap is compacted by all processors in parallel. Graph nodes are locked to maintain sharing and to guarantee correctness. Performance results measured on an eight processor system show that the absolute number of garbage collects hardly increases in comparison to sequential execution. The amount of time spent in garbage collection, however, more than triples because of blocked tasks whose stacks have to be processed as well.

The programmer does not have any direct control on the number of tasks, but the compiler automatically inserts code to decide whether or not to spark new tasks. The decision is based on a crude measure of the system load called 'ForkNow'. At present the runtime support system sets 'ForkNow' to true when the task pool is emptied, and sets it to false when the task pool fills up.

The GAML implementation achieves relative speed-ups between 3.3 and 5.8 for small benchmark programs.

3.2.4 Flagship

The Flagship machine [Watson86, Watson87b, Watson88] builds on experience gained with the Manchester Dataflow project [Gurd85] and the ALICE machine [Darlington81]. Flagship's architecture, however, bears little resemblance to its predecessors since it consists of closely coupled processor memory structures connected through a delta network instead of a shared memory machine with processors connected to memories through a multi-stage network. Although the Flagship machine has a distributed architecture, it provides a single global address space to the application program.

The Flagship machine supports fine-grain parallelism. The program is represented as a collection of *packets*, which may be processed in parallel; a packet is similar to a vector apply node and contains a function with some arguments. Computation is controlled by a set of packet rewrite rules, which are program derived combinators, translated to efficient imperative code. A packet may be rewritten when all of its strict arguments reside in the local packet store. Separate concurrent processes take care of fetching remote arguments and creating local copies; special hardware is included that transparently maps global addresses to locally cached copies for the packet rewrite unit. The parallelism in fetching and rewriting of packets, ensures that communication latency can be tolerated as long as the application contains enough parallelism to keep the processor busy meanwhile; note the resemblance with pipe-lined processors where operands are fetched prior to the instruction execution.

The packet rewriting mechanism as described in [Watson87b] does not use a call stack to evaluate subexpressions since that would give unacceptable high context switch times. Instead the rewritable packet (i.e. redex) is written back to the store as a suspended packet, and new packets are created to evaluate the subexpressions. The storage management in the Flagship machine is thus concerned only with the heap (packet store). Weighted reference counting [Watson87a] is used as the primary means of garbage collection since it provides good intra-processor locality of reference. Occasionally a distributed mark-and-scan algorithm [Derbyshire90] is used to reclaim the cyclic graphs that are left over by the reference counting collector.

To prevent the machine from being flooded with too many fine-grain tasks, the Flagship design includes two task pools per processor: the active packet queue and the holding stack. During ordinary execution rewritable packets are placed in the active packet queue or sent to a remote processor, while the holding stack is used when the machine gets flooded with packets.

The dynamic load balancing of the machine is supported by the delta communication network that propagates load information as well. When a packet is sent into the network for remote execution, it will be routed automatically to the least busy processor, after which the load information is adjusted. Although this approach with feedback balances the load evenly, it does not take locality considerations into account, hence, often remote copying is needed since packets are not routed to the processor which holds their strict arguments. Therefore each packet is sent with a preferred processor number, but the system software overrules the preference when the load distribution gets too uneven.

A multiprocessor emulator consisting of MC68020s interconnected via a custom-designed switching network is under construction.

3.2.5 PAM

The Parallel Abstract Machine (PAM) [Loogen89] is the first of three distributed functional language implementations for Transputer systems. The local memory structure is reflected in the graph reducer, which distinguishes two addressing modes: local and remote. The message handling to support remote pointers is handled by a separate communication unit that operates concurrently with the graph reducer. The communication unit is also responsible for distributing tasks to other processors (load balancing). The reducer and communication unit are implemented as separate Occam processes that are scheduled automatically by the Transputer hardware. The PAM is driven by annotated parallelism in the source program, and the graph reducer hands freshly sparked tasks to the communication unit for scheduling.

The PAM uses the evaluation-transformer model of graph reduction for efficiency, but not for automatic parallelisation: the programmer has to annotate the program source. Each *task* node, which represents a complete function application, includes additional space for a value and a pointer stack, so that the call stack can be implemented as a linked list of activation records, and no special stack management is needed. Pure weighted reference counting is used to reclaim all garbage nodes in the (global) heap; special decrement messages are transmitted to update reference counts on remote processors.

The PAM uses a simple diffusion scheduling strategy to distribute tasks over the machine: whenever a processor is idle, it queries its direct neighbours for work. Although tasks will never be executed more than one hop from their originating processor, references to graph nodes can spread across the entire machine. A static routing scheme is used to transfer messages that fetch remote data. The programmer is solely responsible for controlling the grainsize and number of tasks in the machine; no handles are provided by PAM.

The performance results reported in [Loogen89] are based on a prototype implementation where PAM's abstract machine instructions are interpreted by an Occam program. Since then an improved version of the compiler has been constructed that generates Transputer assembly directly, but no results have been published yet. Good speedups have been measured with the interpreter based implementation, although some benchmark applications suffered from superfluous communication overhead to fetch multiple copies of the same remote graph node.

3.2.6 HDG

The Highly Distributed Graph-reduction (HDG) machine [Kingdon91] is another distributed functional language implementation for Transputer machines, which is also based on the evaluation-transformers graph reduction model. Unlike the PAM, the HDG-machine is based on compiler-derived parallelism and its prototype implementation is more sophisticated.

The graph reducer stack is implemented as a chain of activation frames; the analysis technique of Lester [Lester89b] is used to compute the maximum size of each record in advance. Garbage nodes in the heap are reclaimed with the composite weighted-reference-counting/copying collector [Lester89a], which only uses reference counts for inter-processor references. To support this garbage collector, the sequential graph reducer has been extended with two special node types: *output indirections* that point to remote nodes and carry a weight, and *input indirections* that hold the reference count of a local node. An extra advantage of output indirections is that once an indirection has been overwritten with the remote value, all local copies share that same value. This saves bandwidth in comparison to systems like PAM that do not use output indirections, and have to fetch multiple copies.

The task distribution is regulated by two task pools per processor: The *migratable* pool holds freshly sparked tasks, while the *active* pool holds resumed tasks that have become executable again after waiting for the result of another task. If the active pool is empty then a task from the migratable pool is selected in LIFO-order for execution. If the migratable pool is empty too then a direct neighbour is asked for work. Only tasks from the migratable pool are exported (FIFO order) since their state consists of just one vector apply node. This scheme amounts to diffusion scheduling where tasks may only be executed one hop away from their originating processor. Apart from the LIFO/FIFO selection of migratable tasks, no provisions are made to control parallelism by the runtime support system, despite the programmer's lack of control on the compiler-derived tasks.

A four-node Transputer implementation has been constructed and used for small benchmark programs. The results show that the costs of fine-grain parallelism are rather high: the purely sequential nfib program runs 1.7 times as fast as its parallel counterpart on one processor. The relative speedups are surprisingly good: up to 3.6 on four processors. It remains to be seen whether the fine-grained approach scales to "real" applications on large machines.

3.2.7 PABC

The Parallel ABC (PABC) machine [Nöcker91b] is an abstract machine being designed for parallel graph reduction on distributed memory systems. It builds on the sequential ABC machine [Smetsers91], and is used as an intermediate target machine for implementing Concurrent CLEAN [Nöcker91a]. A prototype implementation for a 64 node Transputer machine is under construction, but not all issues discussed in Section 3.1 have been addressed yet. Therefore, we limit the discussion to some distinguishing features of the PABC machine.

Like most parallel implementations, the spark-and-wait annotation has to be used to denote parallel tasks in the PABC machine. In addition, the programmer has to explicitly control task allocation as well by adding annotations to the source program [Achten91]. The programmer specifies on which processor a new task should run by means of a set of built-in predicates. For example, tasks can be placed at a neighbour, a random processor, or the processor that holds a specific graph node.

For efficiency the graph transport between processors is not on a per node basis, but rather with a sub-graph at a time. Before transmission, the graph is copied into a message buffer, so that it can be transported as a single packet. Maintaining sharing in the transported graph complicates the copy algorithm [van Groningen92].

The parallel graph reducer uses two stacks per task. These stacks are allocated in a fixed-sized block growing in opposite directions. On overflow, the stack pair is reallocated (i.e. copied) to a larger block; this apparent inefficient solution has been proven satisfactory in a simulator. The heap will probably be managed by composite reference-counting/local-copying garbage collector as devised by Lester.

Preliminary performance results for a 16 Transputer implementation have been published [Kesseler92].

3.2.8 GRIP

The GRIP (Graph Reduction In Parallel) machine is a purpose-built shared memory multiprocessor for executing functional programs [Peyton Jones87a, Peyton Jones89, Hammond91]. The hardware consists of up to 20 boards, each holding four processors and one Intelligent Memory Unit (IMU), interconnected by a single fast packet-switched bus (Futurebus). An IMU consists of memory and a microprogrammable data engine that supports graph reduction at a higher level than simple read and write instructions of conventional memory. It performs allocation of variable-sized heap cells, garbage collection, locking of shared nodes, and task scheduling. The IMUs together constitute a uniform accessible shared memory. Each processor (MC68020) is also equipped with a floating point co-processor and a private local memory (1Mbyte) that has to be addressed separately and can not be accessed from outside.

To take full advantage of GRIP's hardware architecture, the graph reducers use part of the local memory attached to each processor as a cache for the global heap that resides in the IMUs. Whenever a global node is accessed, a local copy is created first, and subsequently used. Copying redexes might duplicate work, so the IMU sets a lock bit when a node is fetched for the first time, and automatically attaches tasks to a waiting list on subsequent fetches. When the node is updated, the waiting tasks are placed in the task pool, which is managed by the IMU.

New nodes are allocated in the local memory, not in the IMU. Only when a global node is updated with a local node, a copy of the entire local sub graph is created in the global heap. This *flushing* mechanism prevents the creation of global pointers to local nodes, so that the local garbage can be reclaimed autonomously. When the local heap fills up, a part of the local graph nodes is flushed to the global heap to create new free space. Global garbage is reclaimed by the IMUs in parallel after the reducers have been suspended.

Stacks are allocated in the local heap as (large) fixed sized nodes; if a stack overflows, a new stack segment is allocated and linked to the old one. When the stack shrinks again the new stack segment is discarded, and execution continuous with the old one. The stack of a blocked task is moved to global memory when the processor runs out of local space.

Each processor maintains a local task pool to record expressions that might be evaluated in parallel. When the system load is low enough, some tasks are exported to an IMU while flushing the corresponding expressions to the global heap as well. Based on the system load, which is sampled once per millisecond, the IMUs employ a LIFO (high load) or FIFO (low load) scheduling policy.

To control the excessive generation of parallel tasks, two throttling mechanisms are employed at runtime: 1) the *spark rate* controls the maximum number of tasks a processor is allowed to create in one tick, 2) when the global number of tasks exceeds the *spark cutoff level*, all processors refrain from creating tasks. Preliminary performance results on an 18 processor prototype show that a LIFO scheduling policy in combination with spark cutoff throttling reaches acceptable performance on fine-grained applications: nfib loses only a factor of two in comparison to perfect linear speedup.

3.2.9 HyperM

The HYbrid Parallel Experimental Reduction Machine (HyperM) is the successor to the distributed memory machine developed by the Dutch Parallel Reduction Machine project [Hertzberger89, Barendregt87]. The HyperM architecture [Barendregt92] contains a number of clusters interconnected by a high speed network, where each cluster consists of a few processors connected to a shared memory. The shared memory clusters have been included to increase performance, while retaining the scalability of the original APERM design. The parallel graph reduction inside the clusters is the research topic of this thesis and is discussed fully in Chapter 4; this overview of the HyperM machine includes only the main results.

Unlike the previous parallel machines, the HyperMachine is programmed with a single divide-and-conquer skeleton named *sandwich*. Many applications, however, can be expressed either directly or through transformation as divide-and-conquer programs [Vree90]. To enforce the efficient execution of coarse grain tasks, the sandwich skeleton eagerly reduces all shared data between tasks to normal form before sparking them for parallel execution. This effectively eliminates all problems related to shared redexes as they do not exist. Therefore tasks can be copied safely to remote clusters without duplicating work, and graph nodes in shared memory can always be accessed without locking for exclusive access. A disadvantage of the sandwich is the eager semantics that might lead to non-termination. At the moment the programmer is responsible for assuring that only needed expressions will be evaluated in parallel. This is usually no problem for plain divide-and-conquer algorithms, otherwise the programmer can often transform the non-terminating application into a terminating equivalent program by a set of rules described in Chapter 4.2.

The programmer is required to give a complexity-measure of each task in the sandwich skeleton, for example, the length of the list in the merge sort program. This grainsize information allows the runtime support system to regulate the minimal grainsize according to the system load by inhibiting sparks of tasks that are too small. In addition, work can be classified into two categories: coarse grain tasks that may be allocated at any cluster, and threads that are limited to one cluster and therefore can exploit shared memory. The coarse grain tasks are scheduled on the parallel machine by a hierarchical scheduler [Hofman92a] that uses the grainsize information to determine heuristically how far away a task may be allocated. This has the advantage that work spreads faster over the machine than with distributed scheduling policies like diffusion scheduling, where tasks may only travel one hop.

The sandwich reduction strategy allows for efficient storage management in the HyperMachine. Tasks are copied entirely to remote clusters and do not contain remote pointers, hence, garbage can be collected locally in each cluster. Each thread in a shared memory cluster is provided with a private heap, which is managed by a two-space copying garbage collector independent of other threads [Langendoen92b]. This is possible because the sandwich limits inter thread pointers to child-ancestor pointers; there are no external root pointers into the private heap of a thread. The threads in HyperM time-share one common to-space so only a small fraction of the heap is wasted, not half the heap as in case of general parallel copying garbage collectors.

The tree structure of divide-and-conquer algorithms allows for all tasks on one processor to share one single call stack by stacking their state on top of each other. Only the task on top of the processor stack can execute, but simulation studies have shown that this constraint hardly decreases performance in case of a LIFO/FIFO scheduling policy [Hofman92b]. Thus HyperM's storage manager allocates a single large fixed-sized stack per processor on start up.

At this moment a single cluster consisting of four MC88000 RISC processors and 64Mbyte of shared memory is running, and real speedups over a sequential version have been measured for small benchmark programs like nfib and 8queens.

3.3 Comparison

The discussion of the nine parallel graph reduction machines has been summarised in Table 3.1. Most machines can be classified as either shared memory or as distributed memory machines, except for GRIP and HyperM that combine the two memory types. GRIP is constructed as a shared memory machine, but each processor is equipped with private local memory as well. HyperM on the other hand, is constructed as a distributed memory processor where each processing element consists of a cluster of CPUs connected to a shared memory. The Flagship design is classified as a distributed-memory machine, although it provides a uniform accessible address space through special hardware that transparently maps global data to locally cached copies.

The three Transputer-based distributed-memory machines (PAM, HDG, and PABC) provide a global address space in software through remote pointers, which are interpreted specially by the graph reducers. The distributed memory architecture of HyperM is not visible to the graph reducers since tasks are copied as self contained sub-graphs to remote processors, hence, graph reducers never need to fetch data outside their cluster; inside a cluster the shared memory provides a single global address space. The GRIP-machine is the only shared-memory machine that has a non-uniform address space: the graph reducer distinguishes pointers into its local memory, which can not be accessed by other processors, from pointers into global shared memory. This complicates

	System architecture		
	hardware	**type**	**address space**
$<\nu,G>$	Sequent Symmetry	shared memory	global
AMPGR	BBN Butterfly	shared memory	global
GAML	Sequent Balance	shared memory	global
Flagship	Custom VLSI	distributed memory	global
PAM	Transputers	distributed memory	remote pointers
HDG	T800-25 Transputers	distributed memory	remote pointers
PABC	T800-25 Transputers	distributed memory	remote pointers
GRIP	MC68020s with IMUs	shared (+ local) memory	global (+ local)
HyperM	MC88000(4x) clusters	distr. (+ shared) memory	global (copies)

	Parallelism		
	source	**grainsize control**	**# tasks control**
$<\nu,G>$	spark-and-wait	cut off	fixed sized pool
AMPGR	compiler	in-lining (by compiler)	fixed sized pool
GAML	spark-and-wait	cut off (load info)	fixed sized pool
Flagship	compiler	–	–
PAM	spark-and-wait	cut off	–
HDG	compiler (eval. transf.)	–	–
PABC	low level annotations	cut off	–
GRIP	spark-and-wait	cut off	spark rate + cut off
HyperM	divide&conquer skeleton	cut off	automatic cut off

	Resource management		
	scheduling	**stack**	**heap (garb.coll.)**
$<\nu,G>$	global LIFO	linked frames	concurrent copying
AMPGR	two-level LIFO	save/restore	–
GAML	global LIFO	stack/task	parallel copying
Flagship	diffusion, LIFO/FIFO	–	Weight.RC+mark-scan
PAM	diffusion	linked frames	Weight.RC
HDG	diffusion, LIFO/FIFO	linked frames	copying + Weight.RC
PABC	annotations	stack/task + reallocation	copying + Weight.RC
GRIP	local + global LIFO	linked segments	copying (local+global)
HyperM	hierarchical, LIFO/FIFO	processor stack + ToS	copying per task

Table 3.1: Comparison of parallel graph reduction machines.

the graph reducer since it has to cope (*flush*) local nodes to global memory when updating global nodes.

In general the problems associated with shared redexes are handled by locking to enforce mutual exclusive access, and marking to avoid duplication of work. The HyperMachine is an exception since it employs a reduction strategy that eagerly normalises shared data before sparking tasks. Several designs have taken measures to reduce the locking overheads:

GAML An additional node type has been introduced to denote (potentially) shared application nodes; hence, ordinary application nodes do not need to be locked.

GRIP Locking is performed transparently to the graph reducer by the Intelligent Memory Unit (IMU); hence, only global redexes are locked.

PAM External requests for local data are handled by placing descriptors in the task pool. The single graph reduction unit processes one task at the time, hence, no locking is required at all.

HDG Graph rewrites are made atomic by inhibiting interrupts in critical sections.

The AMPGR, Flagship, and HDG machines are driven by compiler derived parallelism, while the others require programmers to annotate expressions suitable for parallel execution. In the latter case the programmer is also responsible for keeping a minimal grainsize that outweighs the sparking overheads. The AMPGR automatically increases the grainsize of the compiler-derived fine-grain tasks by in-lining the first spark inside every function. The Flagship and HDG machines have no provisions to control the grainsize of their compiler derived parallelism.

To control the number of tasks in the parallel machine, most schedulers employ a LIFO/FIFO allocation policy that achieves good results for divide-and-conquer applications. In addition, the $<\nu,G>$, AMPGR, GAML, and GRIP machines limit the number of tasks by discarding tasks in case of excess parallelism. Tasks may be discarded safely since the parent task will eventually reduce the discarded task itself (evaluate-and-die model).

Scheduling is of particular importance since it is the prime means to control the resource demands and locality in (remote) references of the application. The locality in particular has a large effect on performance, especially in distributed memory machines where the number of messages to resolve inter-processor references should be kept low to avoid communication congestion.

In some shared memory machines (i.e. AMPGR and GRIP) a two-level scheduler is used to achieve locality: tasks are usually placed in a local task pool associated with the sparking processor; when this pool fills up, tasks are spilled over to a global task pool. Diffusion scheduling algorithms, which limit tasks to travel only one hop from their originating processor, are employed by several distributed memory machines. To avoid the slow spreading of work under diffusion scheduling, and enhance locality even further, the HyperMachine uses a hierarchical scheduler; a user-annotated complexity measure indicates at which scheduling level a task may be handled, i.e. how far away the task may be allocated. The PABC machine takes the approach of having the user control the task allocation through program annotations.

The complexity of supporting an arbitrary number of dynamically sized stacks in a parallel graph reduction machine, has resulted in several designs that use a linked list of call frames in the heap instead: $<\nu,G>$, PAM, and HDG. In GRIP a linked list of segments is used; each segment holds a number of stack frames. The PABC machine allocates fixed sized blocks as stack space, and reallocates the stack to a larger block on overflow. Both AMPGR and HyperM use a single stack per processor that is used by all graph reduction tasks allocated to one processor. The AMPGR machine saves a task's state in the heap when it blocks to await the result of another task, when execution resumes the state is restored on the processor stack. HyperM does not save/restore state, but leaves it on the processor stack; the processor stack is used as a stack of stacks, and the scheduler is constrained to schedule only the Top-of-Stack (ToS) task for execution.

Copying garbage collectors are used in most machines because of their efficiency and ability to allocate variable sized nodes. In (small) shared memory machines it is possible to let all processors synchronise when running out of free space, then either all processors participate in copying live data (GAML) or one processor collects garbage while the others continue graph reduction ($<\nu,G>$). To avoid global synchronisations and reduce inter processor traffic, the early distributed memory machines (Flagship and PAM) use weighted reference counting to reclaim garbage. Recent designs (HDG and PABC) combine weighted reference counting to handle inter-processor references with copying collectors to reclaim local garbage fast. The GRIP machine also uses two garbage collectors: plain copying to reclaim garbage in each local memory, and real-time compaction to reclaim garbage in the (global) IMUs. The HyperMachine assigns a private heap to each task, and reclaims garbage for each task individually with a copying collector.

3.3.1 Performance

Performance measurements are necessary to make a quantitative comparison between the parallel graph reduction machines. Ideally a standard benchmark of large 'real' parallel functional programs should be evaluated on each machine, but no such benchmark exists (yet): we have to look at measurements of small toy programs instead. Unfortunately, few results are actually reported for each machine and, worse, different algorithms have been used to solve the same problem. Only the notorious `nfib` program, a one-liner to compute the number of function calls per second, has been coded in similar style and measured on most parallel reduction machines.

machine	processor		sequential	parallel(1)	parallel(#proc)	speed-up
AMPGR	68020	16Mhz		1.3 Knfib/s	19 Knfib/s (15)	– / 15
Flagship						/
PAM	Transputer	20Mhz		1.3 Knfib/s	15 Knfib/s (12)	– / 12
$<\nu,G>$	80386	16Mhz	64 Knfib/s	43 Knfib/s	320 Knfib/s (15)	5 / 8
GRIP	68020	16Mhz	36 Knfib/s	36 Knfib/s	188 Knfib/s (6)	5.2 / 5.2
GAML	NS 32032		19 Knfib/s	12 Knfib/s	69 Knfib/s (8)	3.6 / 5.8
HDG	Transputer	25Mhz	27 Knfib/s	17 Knfib/s	59 Knfib/s (4)	2.2 / 3.5
PABC	Transputer	25Mhz		207 Knfib/s	1795 Knfib/s (16)	– / 8.7
HyperM	MC88000	25Mhz	1520 Knfib/s	1500 Knfib/s	6000 Knfib/s (4)	3.9 / 4.0

Table 3.2: Nfib ratings for various functional language implementations.

Table 3.2 lists the nfib ratings for the surveyed machines as published in various articles. The column labeled 'sequential' presents the results for a pure sequential graph reducer on one processor. The next column shows the ratings for a parallel graph reducer that sparks tasks and incurs other overheads like locking to support parallel execution. Comparing the two columns shows that, except for GRIP and HyperM, parallelism does not come for free: more than 30% overhead costs are incurred. The GRIP machine does not loose performance since the locking actions are performed in parallel by the IMUs, and the hand-annotated nfib program produces just 32 coarse grain tasks so negligible task management overhead is incurred. The HyperM also uses coarse grain tasks, which explains why it only incurs a 1% loss due to task creation overhead.

The column labeled 'parallel(#proc)' presents the maximum nfib rating measured on the machine, the number of processors is included in parenthesis. Finally, the last column provides the real speed-up over the sequential implementation, as well as the relative speed-up over a parallel run on a single processor. Note that only the slowest sequential implementations reach perfect linear speed-up, while others suffer from a loss in efficiency due to task management overhead.

It is impossible to draw any sensible conclusion about important design issues taken in each machine from the absolute nfib ratings of Table 3.2 since the bare hardware performance differs greatly. The relative speed-ups provide no fair comparison either since it is rather easy to speed-up slow sequential implementations, but it is far more difficult to speed-up state-of-the-art graph reduction.

3.4 Conclusions

The lack of a comprehensive benchmark of parallel functional programs and corresponding performance measurements makes it impossible to draw conclusions about the impact of important design decisions like memory allocation policy, task scheduling, etc. Therefore, we conclude by signaling some trends observed in the surveyed parallel machines (see Table 5.2).

Most machines are driven by explicit parallelism though annotations in the program source; only the Flagship and HDG machine exploit implicit parallelism detected by the compiler. The general spark-and-wait annotation is always used in combination with explicit grain size control by the programmer. This trend of having the programmer annotate parallelism and control grainsize contrasts sharply with the initial interest in functional languages: early parallel machines described in [Treleaven82, Kennaway83] all try to exploit the implicit parallelism of functional programs without any user assistance.

The current set of parallel implementations of lazy functional languages, however, do still require less user assistance than their imperative counterparts since task scheduling and storage management are automatically handled by the runtime support system. Each machine uses its own unique set of resource management policies, but in case of heap management most designs include a two-space copying garbage collector adapted for parallel processing.

To catch up on the performance of non-functional competitors like object-oriented based systems, many researchers follow the advice of [Vrancken90] and concentrate on advancing sequential compilation technology. This trend is likely to change future parallel implementations of functional languages since fast implementations are rather sensible to runtime support overhead costs.

Chapter 4

WYBERT: graph reduction on shared memory

Shared-memory multiprocessors are suitable targets for parallel functional language implementations since these architectures support the parallel graph reduction model in hardware. Multiple reducers can straightforwardly rewrite redexes in the shared program graph, provided that application nodes are equipped with locks to avoid two graph reducers rewriting the same redex. This "natural" fit between parallel graph reduction and shared-memory multiprocessors eases the parallel implementation of functional languages in comparison to distributed-memory machines, see Chapter 3, but at the cost of scalability since only a limited number of processors can execute in parallel in a shared memory multiprocessor without saturating the bus to memory. To push the point of saturation as far as possible, the WYBERT approach to parallel graph reduction on shared memory multiprocessors employs the cache local to each processor to its full extent. When the performance needs exceed the capacities of one shared memory multiprocessor, several WYBERT machines can be grouped together to constitute a HyperMachine as described in Section 1.4.

The usage of WYBERT as a building block for a scalable distributed memory HyperMachine shows through in several aspects of the design such as the generation of coarse-grain parallelism through explicit user annotations. Section 4.1 presents an overview of the basic decisions taken to achieve high performance on shared-memory multiprocessors, while the later sections contain in-depth discussions of the most important design aspects of WYBERT: process synchronisation (Section 4.2), task scheduling (Section 4.3), and storage management (Section 4.4). Implementation details and performance results of the integrated WYBERT system will be presented in chapters 5 and 6.

4.1 Design considerations

The rapid improvements in sequential implementation techniques of functional programming languages, a hundredfold increase in execution speed in five years, has prompted for a strict separation between graph reduction and parallelism in WYBERT. This assures that new reduction mechanisms can be incorporated for parallel execution in WYBERT with minimal implementation effort. The software of WYBERT is constructed as a set of graph reducers executing ordinary sequential code that occasionally calls the runtime support system to handle parallel activities like task scheduling. Such a high-level interface at the function call level implies some overhead, but the separation of parallelism and graph reduction has already proven itself in practice: WYBERT started out with an interpreter based on SKI-combinator reduction, and ended up with a state-of-the-art compiler generating code that runs about 30 times as fast.

The WYBERT runtime support system (RTS) is designed for an abstract multiprocessor constructed out of processors with local caches connected to shared memory. Cache considerations are explicitly included in the design of WYBERT because of their importance on the overall performance. For example, for memory bounded applications like graph reduction, caches can effectively reduce the number of requests issued on the global connection to shared memory. Thus, the effective exploitation of caches makes it possible to include more processing elements without saturating the shared memory bottleneck in the multiprocessor.

The performance and portability considerations above, combined with the "external" requirements from the HyperMachine have led to a design that differs considerably from other parallel graph reduction implementations for shared-memory machines as discussed in Chapter 3. The difference is the choice to only support the high-level divide-and-conquer paradigm for generating parallel tasks instead of the more general low-level spark-and-wait model. This divide-and-conquer skeleton named *sandwich* has been taken directly from the preceding APERM prototype machine. The corresponding evaluation mechanism, however, has been refined for usage on shared-memory machines as well, see Section 4.2. The advantage of supporting the divide-and-conquer skeleton is that the RTS can exploit the restricted parallelism to efficiently manage the machine resources.

4.1.1 Divide-and-conquer parallelism

Functional programming languages provide abundant implicit parallelism, but the fine-grain nature does not match with stock hardware. For example, to take full advantage of the caches the grain size of a task should be large enough to overcome the initial cold start misses that fetch the working set into the cache. Since automatic grainsize enlargement is too difficult (see Section 3.1.5), the user must explicitly annotate in the program source the expressions that are worthwhile to be evaluated in parallel. To limit the amount of process synchronisation, as well as for the user's convenience, WYBERT does not support the general low-level fork annotation (spark-and-wait parallelism), but is driven by the *sandwich* skeleton based on the divide-and-conquer paradigm; typical divide-and-conquer programs partition a given problem into parts that can be solved independently of each other, hence, process synchronisation is only required at the beginning/end of a part. The minimal process synchronisation behaviour of coarse-grain divide-and-conquer applications allows for efficient execution on both large distributed memory machines (APERM and HyperM) and shared memory multiprocessors (WYBERT). The programmer is responsible for controlling the grainsize of individual tasks so that management overhead can be tolerated.

A large class of applications can be programmed with straightforward divide-and-conquer parallelism, while transformational methods have been developed to cover synchronous process networks and pipeline parallelism as well [Vree90, Langendoen91a]. Programs with irregular communication patterns like 'the sieve of Eratosthenes', however, can not be handled with the sandwich annotation. The following program shows how, for example, the merge sort algorithm can be expressed as a parallel divide-and-conquer algorithm:

```
psort []   = []
psort [x]  = [x]
psort list = sandwich merge (psort L) (psort R)
             where
                (L,R) = split list
```

When the function `psort` is applied to a list (`list`), that `list` is recursively subdivided into ever smaller lists until the trivial case of the empty or singleton list is met. At each invocation of `psort` the `sandwich` annotation creates two new tasks to sort the left and right halves of the list, which are then combined by the `merge` function into one result. The parallel sort program, however, generates far too many tiny tasks to achieve acceptable performance on a parallel machine. For efficient execution the parallel tasks should satisfy the following constraints:

(a) The result of a task should be needed to compute the final program's result. This condition rules out speculative parallelism, and assures that no processing power is wasted in computing useless values.

(b) The cost to evaluate a task should outweigh the overheads of allocating the task at a remote processor. This guarantees that parallel execution on an idle processor is faster than sequential execution locally.

(c) The task has to be self contained, that is tasks may not share delayed computations that still have to be evaluated (i.e. suspensions/closures). This allows a task to execute independently without any synchronisation with other active tasks.

The programmer can rather easily fulfill conditions (a) and (b), but it is considerably more difficult to meet condition (c) because of the lazy evaluation mechanism that often creates a large number of shared suspended computations. For example, the `psort` function can be improved to generate tasks with a minimal grainsize by switching to the sequential `msort` code when the length of the argument `list` falls below some threshold:

```
threshold = 481

psort []   = []
psort [x]  = [x]
psort list = msort list,            if #list < threshold
             sandwich merge (psort L) (psort R), otherwise
             where
                (L,R) = split list
```

Note, however, that the parallel tasks are not independent since `L` and `R` share the common computation '`split list`'. It is possible to explicitly force the normalisation of `L` and `R` by inserting Miranda's system function `seq`, but this approach is tedious and error prone for large applications. Therefore WYBERT includes a special reduction strategy for the sandwich annotation that automatically normalises task arguments to create independent tasks according to condition (c).

Since this book discusses the implementation issues we do not go into greater detail of how to write parallel programs with the sandwich annotation; the reader is referred to the original work performed within the DPRM project [Vree89], which has shown that the hard part is controlling the grain size, not the insertion of the sandwich annotation.

4.1.2 The FRATS reduction strategy

As already pointed out in Chapter 3 parallel graph reduction is complicated by the existence of shared redexes that have to be updated for efficiency. In general the problem of keeping graph nodes consistent in shared memory is solved by extending the individual nodes with a lock field to enforce mutual exclusive access. In some cases it is possible to encode the lock in the node tag without any space overhead, but the disadvantage of these implicit task synchronisations is the negative impact on runtime performance. Besides the overhead of acquiring and releasing locks, the traffic on the global bus increases since each lock operation has to show through from the local cache to all other caches to guarantee consistency. Measurements reported in Chapter 6 show that applications waste up to 50% of their execution time in locking overheads.

Instead of curing the problem of shared redexes, the FRATS reduction strategy avoids it altogether by adopting APERM's idea of eagerly normalising shared data before sparking parallel tasks [Hartel88a, Vree89]: shared redexes simply do not exist. The sandwich annotation provides an explicit handle to the FRATS (First Reduce Arguments Then Share) reduction strategy to control shared redexes. All potentially shared redexes are "squeezed" out of the tasks by evaluating the function bodies and their corresponding arguments to normal form, as will be explained in Section 4.2. Therefore tasks executing in parallel can only share *read-only* data, hence, tasks may execute independently of each other without locking of graph nodes or any other low-level synchronisations; for distributed systems, tasks can be copied safely to remote processors without duplicating work.

At runtime a divide-and-conquer application programmed with the sandwich annotation (recursively) unfolds into a tree shaped task structure with independently executing leaf tasks. Both the scheduler and storage management of WYBERT take advantage of this regular task structure.

4.1.3 Task scheduling

A straightforward scheduler of "sandwich" tasks on a shared memory multiprocessor uses a global pool of executable tasks where processors store newly created tasks and fetch work when running idle. This straightforward scheduling policy is known as list scheduling [Graham69]. However, to take advantage of the tree-shaped task structure of divide-and-conquer programs, and to avoid the bottle-neck of the global task pool when scaling to large machines, WYBERT uses the LIFO/FIFO scheduling variant of the Manchester throttle mechanism (Section 3.1.5). Each processor maintains a local task pool, where other processors may steal work when running idle. After a short initialisation phase, each processor executes its part of the application's divide-and-conquer tree in depth first order. As a result this scheduling policy exploits the caches very well since the most recently created task is scheduled first, and that task is most likely to find (part of) its data set in the cache.

The LIFO/FIFO scheduler has been adapted to support fast context switching and efficient stack management. The depth-first traversal of the task tree allows all tasks that execute on the same processor to share one reduction stack, the processor stack, as a stack of stacks. At start up, a task sets its private stack pointer to the current top of the processor stack. If the task executes a sandwich and blocks to await the results of its children, the task leaves its local state on the processor stack, and the next fresh task starts to allocate its stack on top of the blocked task, etc. When a blocked task has received the results of all its children, it becomes executable again, but that task may only resume execution after all tasks on top of it have finished, otherwise it could overwrite the state of other tasks.

The Top-of-Stack (ToS) scheduling constraint can lead to a loss of all parallelism in rare cases. Results in Section 4.3, however, show that for a benchmark of divide-and-conquer applications negligible processing power is wasted by idling processors. The advantage of the ToS scheduler is the usage of a single stack per processor, which is easy to manage in comparison to the general problem of supporting a private stack per task whose maximum depth is unknown in advance (see Section 3.1.3).

4.1.4 Local garbage collection

It is already difficult to implement garbage collection correctly for sequential graph reduction, but it is even more difficult for a shared memory multiprocessor that features one global address space and several parallel reducers.

Efficient stop-and-go garbage collection algorithms like mark&scan and two-space-copying traverse the complete graph to identify all live nodes. This forces a total synchronisation of all graph reducers in the multiprocessor because as soon as one graph reducer runs out of free space, all other graph reducers have to be stopped before the (global) collector can safely reclaim the garbage. The interrupted graph reducers have to leave the graph in a consistent state for the garbage collector, which complicates the graph reducer design. For these reasons distributed garbage collection per processor is preferred.

Although the FRATS reduction strategy guarantees read-only access of shared live data, which facilitates local graph reduction, it does not pose any restrictions on accessing garbage! In particular, FRATS does not prohibit that two child tasks both delete a reference to the same shared node, which forces the tasks to synchronise to determine the new status (live or garbage) of that node. As a consequence reference counting algorithms disqualify as local garbage collectors since they require a lock per node in the graph to enforce mutual exclusive access. Local garbage collection based on stop-and-go algorithms, however, is possible for the leaf tasks of the divide-and-conquer task tree.

Leaf tasks can refer to shared data that has been generated by some common ancestor task, but this data will never be updated because it has already been normalised. The lack of updates of shared nodes makes it impossible for tasks to make references to freshly created "local" nodes of other active tasks, hence, pointers between active tasks do not exist. This observation allows for the efficient stop-and-go garbage collection of a leaf task without the need to consult other tasks for incoming pointers. The storage management of WYBERT incorporates local garbage collection as follows:

- At the start of its execution a task is provided with a private heap.
- During execution a task runs a two-space-copying collector (or any other stop-and-go collector) on its private heap to reclaim garbage whenever it runs out of free nodes.
- When a task executes a sandwich it is suspended and its heap may not be garbage collected since (active) offspring can refer to nodes in it.
- At the end of its execution the task's heap is appended to the heap of its parent.

The two-space-copying garbage collection algorithm [Cheney70] has the advantage over the mark&scan algorithm that it can easily handle variable sized nodes, which are commonly used in modern graph reducers.

Besides the advantage of avoiding complex synchronisations between all processors during a global garbage collect, WYBERT's local garbage collection method with private heaps does not need to reserve half of the available memory for the to-space: all processors can time share one common to-space. If the size of a task is bounded to M/p, which is reasonable if memory of size M is to be equally partitioned among the parallel tasks executing on p processors, then the overhead is reduced to a fraction $M/(p+1)$. The restriction that just one processor can collect its garbage at any time does not limit performance much since a single garbage collector already consumes a large fraction of the memory bandwidth.

4.1.5 Evaluation method

The remaining sections of this chapter discus in detail the most important design aspects of WYBERT: The FRATS reduction strategy [Langendoen91b], ToS scheduling [Hofman92b], and local garbage collection [Langendoen92b]. The feasibility of each aspect is assessed individually by simulating the behaviour of a set of benchmark programs; performance results of the integrated WYBERT system will be presented in Chapter 6. The set of benchmark programs used in this chapter is listed in Table 4.1, but not all programs are used in every simulation for historical reasons.

Programs are written in the lazy functional programming language Miranda and annotated with the *sandwich* construct to explicitly denote divide-and-conquer parallelism. The number of lines of source code (without comments and blank lines) is included in Table 4.1 to indicate which programs are "toys" and which are "realistic".

To evaluate the WYBERT system we have built simulation tools and a prototype implementation on real hardware (see Chapter 6). In this chapter the following two simulators are used:

SIS The oldest simulator is based on an extended interpreter of the lazy functional language SASL [Turner79b], which is a predecessor of Miranda. A program is compiled into a set of combinators that includes a special sandwich combinator. The simulator starts interpreting the combinator graph until it reaches a sandwich combinator. Then squeezes the task arguments to normal form, simulates the parallel task execution by sequential evaluation, and registers task specific properties like number of reduction steps, size of task graph, etc. The resulting task description file is used to compute the speed-up on an ideal parallel machine (unlimited

program	#lines	description
NFIB	11	The notorious program that counts the number of function calls needed to compute the 35-th Fibonacci number.
COINS	16	A program that computes all ways in which 2.79 can be paid with coins of value 2.50, 1.00, 0.25, 0.10, 0.05 and 0.01; a list of coins for each possibility is printed.
QUEENS	29	A divide and conquer solution to the 10-queens problem [Langendoen91a].
MSORT	30	Mergesort on a list of 1024 elements (sin 1, ..., sin 1024).
QSORT	24	Quicksort on a list of 1024 elements (sin 1, ..., sin 1024).
DET	43	Computes the determinant of a matrix by straightforward recursive decomposition; a (sub) matrix is represented as a list of lists.
FFT	95	Fast Fourier Transform on a vector of 512 points [Hartel92].
WANG	100	Wang's algorithm for solving a tri-diagonal system of linear equations. The matrix is divided into fixed blocks, the algorithm consists of two parallel passes (elimination and fill-in) on the blocks [Wang81].
15-PUZZLE	109	A branch and bound program to solve the 15-puzzle. The iterative deepening search strategy (IDA*) is used [Glas92].
SCHED	132	A program to find the optimal schedule of a set of tasks on a number of processors. Implemented as a parallel tree search algorithm [Vree89].
COMP-LAB	207	An image processing application that labels all four connected pixels into objects with a unique label [Stout87, Embrechts90].
WAVE	230	A mathematical model of the tides in the North Sea. Consists of a sequence of iterations that updates matrix parts in parallel [Vree89].
RANGE	368	A program to answer a set of queries on a database that is divided in separate parts. Each lookup is performed in parallel [Hartel89].

Table 4.1: The Divide & Conquer benchmark applications

amount of memory, no synchronisation costs, no bus contention, etc.). This SIS (Sandwich In SASL) simulator has been used for evaluating FRATS (Section 4.2) and ToS (Section 4.3).

MiG The second simulator is based on compiled code as generated by the FAST/FCG compiler of Chapter 5. The compiler inserts monitoring code that traces the memory references made by the application. The address trace generation is executed under control of the stripped version of the MiG simulator [Muller93], which is developed to study cache coherency and bus saturation effects of parallel functional programs. The multiprocessor simulator assigns fixed costs to executed instructions, loads, and stores. Semaphore primitives are fully simulated to get realistic synchronisation behaviour. The runtime support system of WYBERT is not part of the MiG simulator and is included as an ordinary part of the application program. The RTS code, however, has been augmented to collect statistics like memory usage of the benchmark programs.

The MiG simulator produces more accurate results than the SIS simulator since it is based on a model of instructions instead of "reduction steps" and because it takes contention on semaphores into account. Even though the MiG simulator does not considers caching effects, it has been measured that the simulator provides accurate execution times for the benchmark programs within a 15% range of the actual measured times on a SUN 4/690.

4.2 FRATS: A parallel reduction strategy

The purpose of the FRATS reduction strategy is to remove the low-level dependencies between parallel tasks annotated by the sandwich construct, so they can be computed independently. This is accomplished by normalising the shared data before sparking the task for parallel execution. The runtime behaviour of a parallel application under FRATS shows a tree of independent tasks that only synchronise at the beginning/end of their execution. This sparse synchronisation structure allows for efficient implementation on shared memory multiprocessors.

The FRATS reduction strategy is a refinement of the sandwich reduction strategy for the APERM distributed memory machine [Hartel88a, Vree89]. An arbitrary expression is sequentially reduced to normal form until an application of the sandwich primitive is encountered.

$\texttt{sandwich G task}_1 \cdots \texttt{task}_n$
where
$\texttt{task}_i = \texttt{F}_i\ \texttt{a}_{i1} \cdots \texttt{a}_{im_i}$
and
$\texttt{F}_i$ and G are arbitrary functions

Then the following steps are taken:

1. All shared expressions are "squeezed" out of the tasks. This means that the function bodies $\texttt{F}_i$ and their corresponding arguments $\texttt{a}_{i1} \cdots \texttt{a}_{im_i}$ are each evaluated to normal form.
2. A set of tasks is sparked to evaluate the arguments of G: $\texttt{task}_1 \cdots \texttt{task}_n$ to normal form and *in parallel*.
3. Upon termination of all tasks from step 2, the function G is invoked with the computed argument values. Then normal order reduction resumes.

The squeeze in step 1 ensures that the tasks sparked in step 2 do not share any redex. Hence, these tasks cannot modify any part of the graph accessible by others. On exit, however, each child task returns its result by updating the corresponding root redex in the parent graph (i.e. an argument of G), but since the parent task is suspended until step 3 this does not cause a consistency problem. The squeeze, combined with suspending the parent, guarantees that active tasks only share read-only data, hence, graph nodes in shared-memory can always be accessed without locking for exclusive access.

The difference between FRATS and the original sandwich reduction strategy for APERM is that expressions at function positions (i.e. the expressions $\texttt{F}_i$) are also reduced to normal form in step 1. APERM can live with some shared redexes at the expense of superfluous work when shared redexes get copied to remote processors. For WYBERT, however, a single shared redex is enough to cause inconsistencies, and cannot be tolerated.

A disadvantage of the FRATS reduction strategy is that the squeeze in step 1 deviates from the standard lazy evaluation mechanism, which might lead to non-termination in the worst case. To avoid any superfluous computation the functions G and $\texttt{F}_i$ from the sandwich definition have to be "extremely" strict in all their arguments. It is not enough to demand strictness in the usual sense of needing a head normal form, since FRATS will completely evaluate those arguments to normal forms to squeeze out all shared redexes. When the sandwich annotation is used with a non-strict function G or $\texttt{F}_i$ then evaluation under FRATS results in evaluation of unneeded expressions, and sometimes non-termination. This problem can be solved by program transformation as will be shown in the next section.

4.2.1 Termination through transformation

In functional programs datastructures like lists are often used as glue between modules that result from functional decomposition. Two modules connected in a producer-consumer relation can communicate via an infinite datastructure because of lazy evaluation semantics as shown in Chapter 2. Such an infinite producer causes problems when it is present in a task argument without the consumer as in the following example:

```
sandwich join (consumer infinite_producer) ....
```

If no special measures are taken, FRATS starts to completely evaluate the datastructure to squeeze out all shared redexes, and will never terminate as the datastructure is infinite. Fortunately a mechanical transformation suffices to change a non-terminating program under FRATS into a terminating one. Without loss of generality we may assume that infinite computations/datastructures are defined by the application of a recursive function to one single data value.

```
infinite    = rec_fun value
rec_fun par = ... rec_fun ...
```

If `infinite` is used by a task in a sandwich construct, then the redex '`rec_fun value`' will be evaluated by FRATS, which results in a non-terminating evaluation. Note that FRATS reduces the arguments of a task separately. This property can be used to prevent the evaluation of an offending redex: the function and value part should be placed in different arguments of the task. This causes FRATS to reduce both parts independently without any problems since the offending redex has been removed. Of course the transformed task has to restore the original redex by applying the function to its argument value during execution.

This solution is called *value-lifting* since the value part of an offending redex will be lifted out as an additional task argument.

> **Value-lifting**: Let `fun` be a function definition that contains an infinite computation denoted by '`rec_fun value`'. Take out `value` as an extra parameter of `fun` and replace all occurrences of

fun with 'fun value'. Repeat lifting until the value appears as a task argument inside a sandwich annotation.

The correctness and termination property of this transformation follow directly from the close correspondence with lambda lifting [Peyton Jones87b]. In the following example the function bin computes a binomial coefficient and uses the sandwich annotation to compute three factorials in parallel. The definition of factorial is based on the equation *facn* $= 1 * 2 * ... * n$; it takes a list of the first n natural numbers and uses the higher order function prod to multiply them.

```
from n    = n : from (n+1)
prod      = foldr (*) 1
nats      = from 1
fac n     = prod (take n nats)
bin n p   = sandwich form (fac n) (fac p) (fac (n-p))
            where
                form fn fp fn_p = fn / (fp * fn_p)
```

The FRATS reduction strategy squeezes all three tasks. In addition to evaluating arguments n, p, and n-p, the function fac will be reduced to normal form as well. This requires processing the function definition of fac, which contains a reference to the list of natural numbers nats. FRATS starts to evaluate the list to normal form since otherwise the three sandwich tasks would use and evaluate elements of nats in parallel. The reduction of nats to normal form never succeeds since the 'from 1' expands into an infinite list. The value-lifting transformation breaks this redex into independent parts by lifting the value 1 (as parameter v) through nats and fac inside the sandwich annotation:

```
nats v   = from v
fac v n  = prod (take n (nats v))
bin n p  = sandwich form (fac 1 n) (fac 1 p) (fac 1 (n-p))
           where
               form fn fp fn_p = fn / (fp * fn_p)
```

Now the transformed program can be safely executed since the squeeze processes the from and 1 as individual components in different task arguments instead of as a redex 'from 1' in a single argument. The evaluation of each task starts by creating its own infinite list of natural numbers by applying the value `1` to function `nats`. As a consequence (part of) the list `nats` will be computed three times, once for each factorial computation. In essence the value-lifting transformation provides each independent task with its own set of infinite datastructures, hence sharing between tasks is impossible. In principle the performance loss could be severe, but the analysis in section 4.2.4 shows that it is negligible for the benchmark programs. As an optimisation, it is not always necessary to perform value-lifting on recursive data: repeating patterns like `ones` below are compiled to finite cycles in the program graph.

```
ones = 1 : ones
```

Cycles won't be unrolled into infinite lists because FRATS records which nodes have already been visited during a squeeze. This also prevents multiple scans of shared data between tasks. After the squeeze the graph cycle does not contain any redexes and can be safely shared between parallel tasks.

4.2.2 Curried functions

The usage of curried functions complicates the recognition of infinite datastructures in the program source because they can generate such expressions at runtime.

```
range a b = take (b-a+1) (from a)
trouble p = .. sandwich foo (.. (range p) ..) ..
```

For example, the function `range` returns the list '`[a,a+1,...,b]`' by taking a prefix of the infinite list '`[a,a+1...]`' as generated by the expression '`from a`'. The definition of range can be seen as two processes connected through a list: a producer part '`from a`' and a consumer part '`take (b-a+1)`'. Evaluation of the term '`trouble 13`' results in FRATS evaluating the curried function '`range 13`' to normal form. In a fully lazy implementation this leads to the instantiation of the producer '`from 13`' because it only depends on the first parameter (`a`) of `range`. The consumer part, of course, cannot proceed without the second parameter (`b`). Hence, FRATS will continue to completely evaluate the infinite list '`[13,14,...]`'.

Curried functions themselves can unfold into infinite datastructures. The previous producer and consumer of `range` can be merged into one function definition:

```
range a b = [],                  if (a>b)
            a : range (a+1) b,   otherwise
```

Although it looks as if `range` does not contain an infinite producer, the term '`range 13`' again represents an infinite datastructure. Note that the subexpression '`range (a+1)`' does not depend on parameter `b` and therefore can be evaluated as soon as parameter `a` is present. This is made explicit by performing fully lazy lambda-lifting [Hughes82] on the `range` definition, which results in:

```
range a         = range0 a (range (a+1))
range0 a next b = [],           if (a>b)
                  a : next b,   otherwise
```

Squeezing all redexes out of the expression '`range 13`' results in an infinite chain of curried functions `range0`:

```
range 13 = range0 13 (range 14)
         = range0 13 (range0 14 (range 15))
         = range0 13 (range0 14 (range0 15 (range 16)))
         = ...
```

As with static infinite datastructures we can use the value-lifting transformation to enforce termination by breaking the dynamically generated redex. With the `range` examples we could lift `p` out of the redex '`range p`' in the definition of `trouble`. For curried functions, however, a simpler transformation is possible.

Order changing

A fundamental observation about the `range` examples is that the consumer part of the definition '`take (a-b+1)`' could not be initiated because it lacked a parameter while the producer of the infinite datastructure did have enough arguments to be evaluated. A simple reversal of the parameters suffices to make the producer dependent on the lacking parameter of the consumer:

```
range a b     = rev_range b a
rev_range b a = take (b-a+1) (from a)
```

Now it is impossible that the term '`range 13`' generates the redex '`from 13`' because it needs the parameter `b` to "call" function `rev_range`. This is due to the underlying semantics of functional languages. Again sharing is lost, but this transformation does not suffer the performance loss of dragging an extra parameter around as with the value-lifting transformation. To minimise loss of sharing we should not modify the general definition of `range` but just the calls that cause non-termination of FRATS's squeeze phase. This can easily be accomplished by inserting the higher order function `delay` at those places in the program source:

```
delay f a b    = converse f b a
converse f b a = f a b
```

The function `delay` will only call `f` when all arguments are present. Hence, the usage of `delay` with the `range` examples will prevent the squeeze from evaluating '`range p`':

```
trouble p = .. sandwich foo (.. (delay range p) ..) ..
```

Cycle naming

As with ordinary datastructures, the squeeze of a curried function can result in an infinite chain of one repeated curried function: a partial application of the same function and arguments. For example, the evaluation of the higher order function `map` applied to one argument.

```
map f []    = []
map f (h:t) = f h : map f t
```

Again we will perform fully lazy lambda lifting for clarity:

```
map f              = map0 f (map f)

map0 f next []     = []
map0 f next (h:t)  = f h : next t
```

The squeeze of the expression '`map sqrt`' will result in an infinite repeating chain:

```
map sqrt = map0 sqrt (map sqrt)
         = map0 sqrt (map0 sqrt (map sqrt))
         = map0 sqrt (map0 sqrt (map0 sqrt (map sqrt)))
         = ...
```

When this chain is represented as a cycle in the graph, FRATS's squeeze does terminate and no program transformation is necessary, just as with the `ones` example. In general, however, compilers do not generate code to create a cycle, but code to build a fresh node with the same curried application. This requires a program transformation to stop FRATS from endlessly building new partial applications. Unfortunately, a little help from the programmer is needed to get the desired cycle in the graph: explicitly naming the cycle through a local function definition suffices. The following definition of `map` forces the compiler to generate code that constructs a cycle at runtime. An extra advantage is that the local function `mf` has one parameter less than the original `map`, which results in fewer reduction steps.

```
map f = mf
        where
            mf []    = []
            mf (h:t) = f h : mf t
```

The class of cyclic unfolding functions is relatively large because in functional programs functions often carry some global state around in parameters, which rarely changes.

4.2.3 Transformation methodology

Whenever the FRATS reduction strategy causes problems, either superfluous computation or non-termination, the programmer has to apply one of the transformations described before. Value-lifting is the most general transformation and can always be applied, but it is also the most drastic one because usually a large number of function definitions have to be changed to lift the "value" to sandwich-level. The other two transformations operate on a single function definition, but can only be applied in a limited number of cases. In general the programmer should proceed in the following way:

1. Locate the offending redex (say `R`).
2. Determine if `R` is a curried function (say '`fun val`')

   yes If `fun` makes a direct call to itself with an unchanged parameter (`val`) then perform Cycle-naming else perform Order-changing on `fun`'s definition.

   no Apply Value-lifting on `R`.

The difficult part is finding the redex `R` in the first step, the rest can be done automatically by some software tool. At the moment, however, all transformations have to be applied by hand.

4.2.4 Performance consequences

In general FRATS's eager reduction strategy, to squeeze out shared redexes, will result in superfluous computation when a task is not strict in all its arguments. This requires a modification of the program to delay the computation by applying a transformation (value-lifting, order-changing, or cycle-naming) from section 4.2.1. If several tasks share a computation that needs to be transformed, sharing will be lost since the transformation causes each task to compute a private version during execution. To quantify the performance consequences of FRATS, superfluous computation and loss of sharing, we have analysed six programs of the parallel functional benchmark suite (Table 4.1): QSORT, FFT, WANG, SCHED, WAVE, and RANGE.

	total amount of execution [reduction steps]			
program	lazy	APERM	FRATS	transf+FRATS
QSORT	558,387	558,429	558,408	426,320
FFT	437,197	441,275	∞	423,674
WANG	121,273	121,524	∞	121,166
SCHED	191,934	194,773	207,455	198,706
WAVE	236,362	238,637	236,603	231,630
RANGE	9,871,499	10,640,802	∞	7,789,470

Table 4.2: Benchmark results of SIS simulator

The first run of the benchmark with the SIS-simulator of Section 4.1.5 was performed without any squeezing of arguments to measure the pure run length of the programs. The results are listed in the column labeled "lazy" of

program	transformations
QSORT	cycle-naming (2×)
FFT	cycle-naming (2×)
WANG	cycle-naming (2×)
SCHED	order-changing (1×)
WAVE	cycle-naming (2×)
RANGE	cycle-naming (5×) + value-lifting (1×)

Table 4.3: Applied transformations

Table 4.2. These values will be used as a reference to derive the amount of superfluous computation encountered by the other reduction strategies. The next column labeled "APERM" contains the results of using the APERM reduction strategy. Comparison with the first column shows that only the RANGE program incurs non-negligible superfluous computation (8%). The FRATS reduction strategy is more strict than APERM since it also reduces the expressions at the function position of a task. This shows in the third column in table 4.2 where three applications fail to terminate under FRATS. The SCHED program takes considerably more reduction steps, whereas the other two need somewhat fewer steps than under APERM. This last decrease is caused by a small optimisation in FRATS that only squeezes $n-1$ sandwich arguments, whereas APERM processes all arguments.

Next the cycle-naming transformation was applied to each benchmark program. As a result all transformed programs do terminate under FRATS. In addition SCHED and RANGE needed a value-lifting and an order-changing transformation respectively to limit superfluous computations. Especially RANGE was sensitive to superfluous computations because the queries did not cover the whole database, which would be completely evaluated by FRATS without any transformation. The applied transformations are listed in Table 4.3, and it shows that only a small number was needed.

The final performance of the transformed programs is listed in the last column of Table 4.2 labeled "transf+FRATS". A remarkable observation is that all programs except SCHED require fewer reduction steps than the original version. This is due to the cycle-naming transformation which uses a local function with one parameter less. The implementation of SASL (bracket abstraction, see [Turner79a]) is very sensitive to the number of parameters: worst case execution time is exponentially proportional to the number of parameters. It is also the cause of the decreased performance of SCHED: order-changing was

	speed-up		
program	lazy	APERM	FRATS
QSORT	3.58	3.51	4.81
FFT	4.55	3.83	4.09
WANG	4.46	4.07	4.14
SCHED	12.14	9.76	11.23
WAVE	1.85	1.82	1.84
RANGE	3.91	1.90	1.85

Table 4.4: Parallel performance compared to column "lazy" in table 4.2

applied to a function with six parameters whose last one needed to be swapped with the first. This parameter shuffle is completely responsible for the incurred overhead.

Although the benchmark results show that a few transformations suffice to avoid unnecessary computations, the squeeze of shared data by FRATS might severely reduce the parallelism of an application. Therefore we have looked at the speed-ups on an ideal parallel machine (∞ processors, no task set-up time) as computed by the SIS-simulator. Table 4.4 contains these computed optimistic parallel speed-ups for the benchmark under various reduction strategies. It shows that FRATS performs slightly better than APERM in most cases and approaches the ideal values in the "lazy"-column. In case of QSORT FRATS even outperforms the original program because of the cycle-naming transformation. The disappointing performance of the RANGE application is a simple loss of parallelism caused by the normalisation of the database before the parallel queries. Further research is needed to improve FRATS performance in this case.

The results of the benchmark programs as measured by the SIS simulator show that with a few program transformations tasks can be made independently of each other so locking at the level of graph-reduction is unnecessary. Eliminating shared redexes in advance only causes minor superfluous computation and does not significantly serialise execution (i.e. lower speed-ups), hence, all benefits are for free. The following sections will discuss two high-level optimisations that can be applied in scheduling and in memory management because of the task independence as enforced by FRATS.

4.3 Top-of-Stack scheduling†

A straightforward scheduler of "sandwich" tasks on a shared memory multiprocessor uses a global pool of executable tasks where processors store newly created tasks and fetch work when running idle. This straightforward scheduling policy is known as list scheduling [Graham69]. Although applications under list scheduling never execute more than twice as long as under an optimal scheduler, list scheduling has two practical disadvantages. First, the global pool eventually becomes a bottle-neck when scaling to large machines; the Buckwheat implementation already suffered from contention conflicts with 8 processors [Goldberg88b]. Second, an idle processor always fetches a task from the global pool without considering whether the task is large enough to outweigh the communication costs: (part of) the task's data has to be transferred from the cache of the processor that created it to the cache of the idle processor. Currently, the latter disadvantage is of no concern for WYBERT since the programmer is required to enforce a minimal grain size of each task (sandwich constraint (b) on page 80).

WYBERT is solely based on divide-and-conquer parallelism, so we can use the local-LIFO/steal-FIFO scheduling variant of the Manchester throttle mechanism (Section 3.1.5) where each processor maintains a local task pool. Processors schedule their local tasks in LIFO order, but steal a task from another processor's pool in FIFO order when running idle. After a short initialisation phase, each processor executes its part of the application's divide-and-conquer tree in depth first order. This scheduling policy exploits the caches very well since the most recently created task is scheduled first, and that task is most likely to find (part of) its data set in the cache.

Whenever a task executes a sandwich primitive to spark new tasks for (parallel) execution, this task becomes blocked until all its children have finished. While the parent task is blocked the associated processor is used to process some other task ready for execution. Hence, a sandwich primitive causes a context switch in the processor: the state of the current task has to be saved, and the state of the new task has to be loaded. To minimise context-switch time the graph reducer usually allocates a private stack for each task so that only the top-of-stack pointer needs to be saved/restored instead of the complete contents. The price for this optimisation is that an arbitrary number of stacks has to be accommodated instead of a single stack per processor; a single stack is much cheaper to manage, see Section 3.1.3 for details.

†This section represents joint work with Rutger Hofman.

The LIFO/FIFO scheduler described above has been adapted to combine a fast context switch time with the advantage of a single stack per processor; the depth-first traversal of the task tree allows all tasks that execute on the same processor to share one reduction stack, the processor stack, as a stack of stacks. At start up, a task sets its private stack pointer to the current top of the processor stack. If the task executes a sandwich and blocks to await the results of its children, the task leaves its local state on the processor stack, and the next fresh task starts to allocate its stack on top of the blocked task, etc. When a blocked task has received the results of all its children, it unblocks and can resume execution. An unblocked task, however, may only be selected for execution after all tasks on top of it have finished, otherwise it could overwrite the state of other tasks.

The ToS scheduling policy of WYBERT maintains on each processor a list of tasks ready for execution and a stack of blocked tasks. Whenever the scheduler is requested to select a new task for execution, it first checks whether or not the blocked task on top of the local stack has become ready for execution. If the topmost task is ready then it is selected to resume execution, otherwise the local task list is inspected. If the list is non-empty then the task in front is selected for execution (LIFO policy), otherwise the ToS scheduler inspects the pools of the other processors in a cyclic manner until it has found a non-empty task list. It steals the last task in that list (FIFO policy) and returns the task for execution at the local processor. Sparking a task amounts to simply inserting the fresh task in front of the local task list.

Although the ToS scheduler does not support task preemption nor task migration for efficiency, the scheduling policy is deadlock free for the following reasoning. Suppose a parallel divide-and-conquer application under ToS reaches a deadlock situation. Let T be the youngest of all (blocked) tasks, that is, T started execution after all other tasks. Since T is the youngest task it can not be waiting for an even younger child task, hence, T is a runnable task blocked by some other task $T1$ lying above it on the same processor stack. This implies, however, that $T1$ started execution after T, which contradicts the assumption that T was the youngest task, hence, deadlock is not possible under ToS. Thus, the ToS constraint can only cause poor performance as is shown in the next section.

4.3.1 Worst case behaviour

A list scheduling (LS) policy obeys the constraint that idle processors and executable tasks do not coexist. This means, that a processor that finishes its task must immediately select another from the (global) list of executable tasks. The policies in LS differ in their selection criterion from such a list. All LS policies have a good performance [Graham69]; the parallel execution time never exceeds twice the optimal execution time if communication delays are neglected. This bound holds for arbitrary precedence relations between tasks and arbitrary execution times of the tasks. In practice list schedulers perform much better on average than the factor two worst case bound.

Distributing the global task pool of LS by equipping each processor with a local pool preserves the valuable list scheduling property; when a processor runs out of local tasks, it starts polling the others to find an executable task, hence, no processor will run idle as long as there is an executable task somewhere in a local task pool. To minimise memory usage, a processor manages its local pool in LIFO order, but tasks are stolen in FIFO order from a remote pool to minimise data communication. Since both this LIFO/FIFO scheduling policy and LS preserve the list scheduling property, they do not differ from a theoretical point of view. Of course, the improved locality in the schedules gives better cache hit rates for LIFO/FIFO than for pure LS, hence, absolute performance is increased.

The Top-of-Stack (ToS) scheduler of WYBERT violates the list scheduling property since it restricts the scheduling of re-awakened tasks, i.e. tasks that have become executable again after all children have terminated. The state of suspended tasks is stacked on top of each other, so at any time only the top-most task may be selected for execution. If the top most task still awaits the result of some child task, it effectively blocks the execution of all tasks below. As a consequence some processor might run idle in presence of an executable task because that task is not on top of a processor stack. This situation is illustrated by the following simple example.

We use a synthetic application that executes two sandwich primitives to create the task structure depicted in Figure 4.1. All task execution times are 1 step except for task 4, which takes 4 steps. In Figure 4.2 we show a possible ToS schedule on two processors. This task schedule takes 8 steps under ToS, while the equivalent LS schedule takes 7 steps; join task 7 has to continue execution on processor 1 because its corresponding fork task 2 was scheduled on processor 1, hence, the execution of task 4 blocks the resumption of join task 7 until t = 6, while processor 2 runs idle.

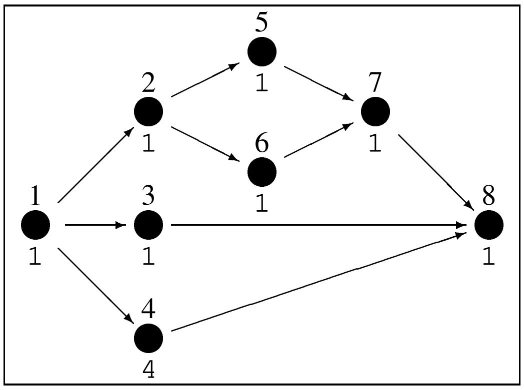


Figure 4.1: A task graph. Tasks are labeled with a task number (above) and an execution time (below the task node).

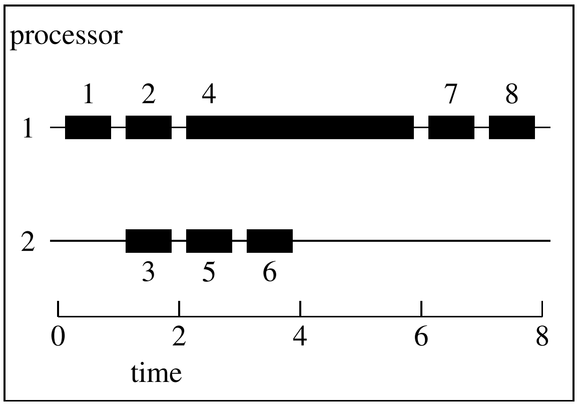


Figure 4.2: A ToS schedule of the task graph of Figure 4.1.

The performance degradation in this example is not significant, but a worst-case task graph can be constructed where all parallelism is lost. On a p processor machine, it can be arranged to let an application deposit p large join tasks at a single processor that have to be executed sequentially because of the ToS constraint, while an ordinary LS policy could schedule these p tasks to run in parallel. Details of this contrived worst case application can be found in [Hofman93]; here we will only give the ratio of execution times of ToS (ω) and LS (ω_0) for brevity:

$$\frac{\omega}{\omega_0} = \frac{pt_{join} + 2p + 2}{t_{join} + 2p + 2}$$

The ratio approaches p for large join tasks ($t_{join} \rightarrow \infty$), which means practically all parallelism is lost, irrespective of the number of processors.

4.3.2 Performance from Simulation Studies

The performance consequences for the scheduling worst case caused by ToS are horrendous. However, there may be a notable discrepancy between worst case performance and "practical" performance. Therefore we have conducted a number of simulations to evaluate the practical consequences of ToS. The performance simulator takes a task graph description and models the execution under a specific scheduling policy on a range of shared memory multiprocessors: with 2, 4, 8, and 16 processors. The supported scheduling policies are LS, LIFO/FIFO, ToS, and Gl-ToS (Global-ToS). Gl-ToS, like LIFO/FIFO, is an intermediate scheduling discipline between LS and ToS: it is the combination of a global task list and stack per processor (ToS constraint). The distinguishing properties of the four scheduling are tabulated below:

	global task list	local task lists
stack per task	LS	LIFO/FIFO
stack per processor	Gl-ToS	ToS

The performance simulator does not take locality effects into account so we do not expect a large difference between LS and LIFO/FIFO.

The Divide & Conquer applications benchmark

To start with, we will evaluate the performance consequences for the kind of applications WYBERT is designed for: parallel Divide & Conquer algorithms. We have made use of the SIS simulator and the following benchmark applications: COINS, QUEENS, MSORT, FFT, WANG, SCHED, and COMP-LAB, see Section 4.1.5.

Some applications are "toy" programs, others are "real world" programs. The corresponding characteristics of the task graph descriptions are listed in Table 4.5; time is expressed in the number of reduction steps, i.e. the number of executed Turner combinators. Average parallelism is the maximum speed-up that can be achieved with an unlimited number of processors [Eager89]. Note

	average parallelism [Eager89]	sequential steps × 1000	average fork task steps	average mid task steps	average join task steps
COINS	246.1	53,643	339	39,903	60
QUEENS	82.2	2,106	73	1,295	10
MSORT	5.0	455	755	1,240	1,579
FFT	5.3	466	3,666	3,424	255
WANG	16.7	5,039	1,275	38,107	1,235
SCHED	56.8	2,188	1,672	7,014	3
COMP-LAB	19.2	169,399	48,264	22,538	263,932

Table 4.5: Properties of the applications of the Divide & Conquer benchmark

that MSORT and, notably, COMP-LAB are the applications that have non-negligible work in the join tasks, so these applications may cause higher ToS degradation than the others.

For efficiency tasks are created only when their grain size exceeds some application-specific threshold. The conditional forking has been explicitly indicated by the programmer. The threshold value is chosen such that the computational demand of leaf tasks considerably exceeds the overhead for task creation (which is set to 250 steps). As a typical example, in Table 4.6 we show the speed-up obtained for the various applications on an 8-node shared memory machine.

Inspection of the speed-up figures on all machines shows that LIFO/FIFO, LS, and ToS yield similar performance, while Gl-ToS tends do less well for most applications; in case of COMP-LAB Gl-ToS performs on average 28% worse than plain LS. This corresponds to our intuitive remark that large execution times of join tasks, as is the case for COMP-LAB can seriously affect the overall performance. The reason that ToS does not suffer from scheduling constraints with the COMP-LAB application can be explained as follows. ToS degradation occurs when executing tasks block resumption of otherwise executable join tasks. This means that the blocker is not a descendant of the blocked join task, because otherwise it would not be executable. Therefore it is necessary for ToS blocking that a task from another fork-join subgraph is allocated at the processor under consideration. Local task lists favour execution of complete subgraphs: processors put newly created tasks into their local task list, and whenever they finish their current task, they first look in their local list for a new task. In this manner, processors have a strong preference for execution

	LS	LIFO/FIFO	ToS	Gl-ToS
COINS	7.7	7.8	7.8	7.5
QUEENS	7.1	7.1	7.1	7.0
MSORT	3.5	3.5	3.5	3.3
FFT	3.7	3.7	3.7	3.7
WANG	6.2	6.0	6.0	6.1
SCHED	7.1	7.1	7.2	7.0
COMP-LAB	7.3	7.3	7.3	4.0

Table 4.6: Speed-ups on the 8-node shared memory machine

of their own offspring. LIFO management of the private task list causes a depth-first traversal of the task subgraph springing from the current task, so tasks will be evaluated together with all their offspring by one processor unless some stealing occurs. Tasks are stolen by another processor only when such a processor runs out of work: it has either completed its part of the task graph, or the join tasks it owns are all blocked. The task that is stolen, is the least recently created task (in other words, stealing is done in FIFO manner). Typical divide-and-conquer applications repeatedly decompose a problem into smaller sub problems, hence, chances are high that the stolen old task represents a considerable amount of work since the "local" task tree is traversed in depth-first order. This means that the stealing processor will be satisfied for a long time, so FIFO stealing lowers the number of allocations to other processors. These aspects, local task lists, LIFO list management and FIFO stealing, work together to limit allocation to another processor, which reduces the chance of blocking executable tasks.

We want to compare the performance of the task list policies, independent of the number of processors and application. Therefore the speed-up figures were geometrically averaged (see [Fleming86]) over the architectures. The comparison shows that LS, LIFO/FIFO, and ToS achieve equal speed-ups, while Gl-ToS performs 5% less than the others. Thus although the ToS constraint can reduce performance (cf. Gl-ToS), the local task lists effectively prevent blocking tasks from degrading performance.

Synthesised fork-join task graphs

We found that most Divide & Conquer applications in the benchmark are suitable for ToS scheduling because their join tasks are small in execution steps, or the applications have a very regular task graph: the exception is COMP-LAB but only for the Gl-ToS scheduling strategy. This does not give much information about the applications for which ToS performs poorly. The following properties induce ToS degradation:

- non-negligible execution times of the join tasks
- irregular task structure (it must be a fork-join graph, of course, but not entirely symmetrical in its forking)
- irregular join task execution times

Finding enough applications that have these properties is hard. Therefore, we synthesised 70 task graphs using random generators

- for deciding whether a (non-join) task will fork or not
- for determining the execution time of the join tasks; these times follow a uniform distribution to obtain a large spreading

A description of the synthesised task graphs is given in Table 4.7.

Class	average number of tasks	average join task steps	average parallelism	sequential steps $\times$ 1000
I	1400 ± 2100	980 ± 80	30 ± 40	1400 ± 2100
II	1400 ± 1800	4990 ± 270	22 ± 25	3000 ± 4000
III	1600 ± 2200	25100 ± 2900	23 ± 28	14000 ± 20000

Table 4.7: Characteristics of the synthesised tasks. Values were drawn from several uniform distributions, so spreading is high. The fork tasks and leaf tasks all take 1000 steps.

The average execution time of the join tasks is either equal to the average execution steps of fork tasks and leaf tasks (class I), or 5 times this average (class II), or 25 times this average (class III). COMP-LAB, the one application that (sometimes) suffers from ToS degradation, has such a ratio of 8.4. Inside each synthesised class the applications differ in the total number of tasks. Applications where join tasks execute on average for 5 times as long as fork tasks and leaf tasks are rare; applications where the join tasks take 25 times as long are even less likely. The reason we included them, is because we expect to find performance degradation for ToS for this type of application if it is to be found at all except by careful construction, as we did for our worst case performance example.

The applications were run on the same architectures as the Divide & Conquer benchmark. Performance of the policies is derived from the speed-up figures in the same way: geometrically averaged speed-ups were calculated per architecture and per application, and performance figures are normalised to LS=1. The applications are collected into three groups, distinguished by different average join task steps. The performance figures we present in Table 4.8 are averages for these three groups.

	LS	LIFO/FIFO	ToS	Gl-ToS
I	1	1.00	0.99	0.91
II	1	0.99	0.97	0.80
III	1	1.00	0.99	0.71

Table 4.8: Averaged speed-ups relative to LS for the synthesised tasks.

The expectations on ToS behaviour prove to be correct for the Gl-ToS scheduling policy. The applications of class I, where join tasks on average take as many execution steps as fork tasks and leaf tasks, show a performance degradation of 9%. Compare this to the application MSORT, where the join tasks also on average take as long as leaf tasks. The essential difference must be the irregularity of execution times of the join tasks. For the applications of class II, the degradation is worse: 19%. For the applications of class III, the performance difference mounts to 29% for Gl-ToS. This performance degradation, however, is not as bad as one would expect; such a virtually random allocation of join tasks to processors might as well result in performance close to the worst case.

Again the ToS strategy, which uses local task lists per processor, performs much better than the global Gl-ToS policy and only suffers a minor performance degradation for all application classes in comparison to LS and LIFO/FIFO: less than2%. This is in accordance with our findings from the Divide & Conquer benchmark.

The results in this section have shown that the top-of-stack constraint does not lead to performance degradation in practice. The simulated execution of the benchmark programs do not show any performance degradation for ToS in comparison to LS and LIFO/FIFO schedulers. Even the synthesised applications with large join tasks do not show any degradation that approaches the worst case of loosing all parallelism. These results that divide-and-conquer applications can make efficient use of a single stack per processor, while exploiting the caches in the multiprocessor by traversing local parts of the task tree in depth-first order.

4.4 Memory management for parallel tasks[†]

An important property of logic, object-oriented, functional, and other high-level programming languages is their automatic management of dynamically allocated storage. The language support system provides the user with a virtually unlimited amount of storage by running a garbage collector to reclaim storage that is no longer in use. The efficiency of the garbage collector is important for the application's performance, especially when the underlying computational model (e.g., graph reduction) often allocates small pieces of memory that are used only for a short time.

From the three classes of garbage collection algorithms (reference counting, mark&scan, and copying collectors), the copying collectors perform best on systems with considerably more memory than the amount of live data [Hartel90]. There are two reasons for the better performance: 1) they only traverse live data, which usually accounts for only a small fraction of the total heap space, while mark&scan collectors access every heap cell twice, 2) copying collectors compact the live data into one consecutive block, which facilitates the fast allocation of (variable sized) nodes by advancing the free pointer instead of manipulating a linked list of free cells and managing the reference counts.

Cheney's two-space copying collection algorithm [Cheney70] is the basis of many (parallel) copying garbage collectors. The available heap space is divided into two equal parts: the from-space and the to-space. During normal computation new nodes are allocated in from-space by advancing the free-space pointer through the from-space. When the heap space in the from-space has been consumed, all live nodes are evacuated (i.e. copied) to the empty to-space by the garbage collector.

global data	from-space	to-space
	↑ *flip* ↓	
global data	to-space	from-space

Figure 4.3: Memory layout for two-space collector

The evacuation starts with copying the nodes in from-space that are referenced by root pointers in the global data area, which contains for example the call stack. Then the nodes in to-space are scanned for pointers to objects in from-space that still have to be copied. This process is repeated until the

[†]This section represents joint work with Henk Muller.

to-space contains no more references to nodes in from-space. The strict separation of global data and the heap allows the collector to efficiently detect with one compare instruction whether a pointer refers to a node in from-space or not. After evacuating all live nodes, the roles of the two semi spaces are *flipped*, and ordinary computation is resumed.

A straightforward adaptation of a copying collector to run on a multiprocessor is to let all processors participate in a global evacuation operation: processors allocate large blocks of storage in the shared global heap, and if one processor detects the exhaustion of the (global) from-space, it synchronises with the other processors to start garbage collection. The evacuation of live nodes proceeds with all processors scanning parts (pages) of the to-space in parallel. To handle possibly shared data objects, processors lock each individual node in from-space when inspecting its status and, if necessary, copying it to to-space. This method is, for example, used in MultiLisp [Halstead Jr84] and GAML [Maranget91].

To reduce the locking overhead of the above method, the Parlog implementation described in [Crammond88] partitions the heap among the processors, so that each processor can collect its own part of the heap. Whenever a processor handles a *remote* pointer to a live node in another part of the heap, it places a reference to the pointer in the corresponding processor's Indirect Pointer Stack (IPS). After a plain evacuation operation, each processor scans its IPS buffer, which contains (new) roots into its private heap, updates the pointers to point to copies in to-space, and continues with scanning the new objects in to-space. Now only the IPSes have to be guarded with locks instead of each heap object.

A rather different approach to use copying collectors on parallel multiprocessors is described in [Appel88]: one processor reclaims all the garbage, while the others proceed with their normal computational work. The synchronisation between the collector and the other processors (mutators) is accomplished through standard hardware for virtual memory. When the evacuation of live nodes starts, the collector copies all root nodes to the to-space, and marks the virtual memory pages of the to-space as inaccessible to the mutators. Then the mutators immediately resume execution in the to-space, while the collector scans the to-space page by page for references to nodes in from-space that still have to be evacuated. Whenever the collector has finished a page of the to-space, it makes that page accessible to the mutators. If a mutator tries to access an object in a not-yet-scanned page in to-space, the hardware generates an access violation trap. This triggers the collector to handle the referenced page immediately, after which the mutator resumes execution.

A common disadvantage of the above copying garbage collection algorithms for multiprocessors are that they waste half of the shared heap, which is reserved for the to-space, and that they require global synchronisation operations. The inherently global nature of these algorithms also raises efficiency problems when scaling to large (hierarchical) shared-memory multiprocessors: the single virtual memory collector cannot keep up with many mutators, while the parallel scan of the other algorithms overloads the memory bandwidth.

4.4.1 Local copying garbage collection

The WYBERT scheme for copying garbage collectors on shared-memory multiprocessors provides each parallel task with its own heap and performs garbage collection per task locally without any global synchronisation with other tasks or processors. This approach is attractive since it avoids global synchronisation and cooperation of processors, while the reserved amount of to-space can be reduced by limiting the maximum heap size of a task and time-sharing a common to-space. Collecting a task, however, requires access to all global root pointers into the local heap. Recording all roots pointing from outside into the heap of some task is a space and compute intensive task in general, especially when tasks can exchange arbitrary data including heap pointers. This makes the scheme of collecting garbage per task unattractive for general parallel processing since each and every communication has to be checked for cross pointers.

The fork-join task structure of our divide-and-conquer parallelism allows efficient incorporation of the above local copying collector scheme in a shared-memory multiprocessor. At runtime a divide-and-conquer application (recursively) unfolds into a tree shaped task structure, see Figure 4.4(a). Each task is provided with a "private" part of the shared heap where it allocates storage during its execution. Interior tasks (1, 2, and 3) are suspended during the execution of their child tasks, so only leaf tasks (4, 5, 6, and 7) can reclaim their garbage locally.

The garbage collection of a leaf task with a two-space copying collector requires the allocation of a contiguous to-space and access to all *root* pointers into the private heap. The latter requirement is hard to fulfil in general, but the divide-and-conquer model causes the leaf tasks to execute without any external interaction, hence, a leaf task cannot pass a pointer to any other active task; there are no pointers between tasks 4 and 5 in Figure 4.4(b). The absence of communication between leaf tasks, however, does not rule out data sharing

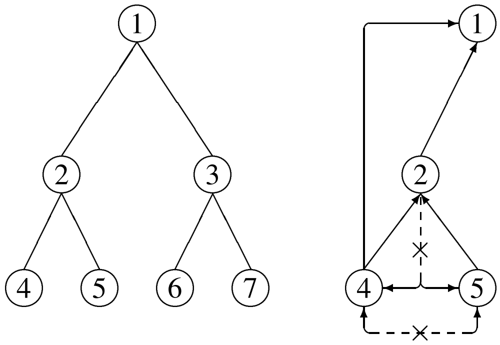


Figure 4.4: fork-join tree (a) with limited inter-task pointers (b).

since tasks can execute different subproblems that contain pointers to shared data in common ancestor heaps. For example, tasks 4 and 5 can share data that resides in the heap of task 2, or even in task 1. Since the FRATS reduction strategy normalises the shared data in advance, that data is read-only and will not be updated, hence, tasks cannot pass pointers through their ancestor's heap. As a consequence pointers from interior tasks to leaf tasks do not exist; for example, there are no pointers from task 2 to either task 4 or task 5 as shown in Figure 4.4b.

Since the divide-and-conquer paradigm limits the inter-task pointers to references to ancestor data, there are no "external" root pointers into the heap of a leaf task. This allows the garbage of a leaf task to be reclaimed with a local sequential copying collector, which only scans the task's call stack for root pointers. Note the resemblance with generation scavenging garbage collectors [Liebermann83] where often the youngest generation (cf leaf tasks) is collected, but not the older generations (cf interior tasks).

Scattered heaps

To accommodate an arbitrary number of tasks, heap memory is allocated in variable sized blocks. Whenever a task runs out of memory it invokes the garbage collector to reclaim space no longer in use. When the amount of live data approaches the blocksize the garbage collector allocates a block twice the current size on the next collection. This assures that a private heap is always a single block, so an ordinary sequential two-space copying collector can be used for leaf tasks.

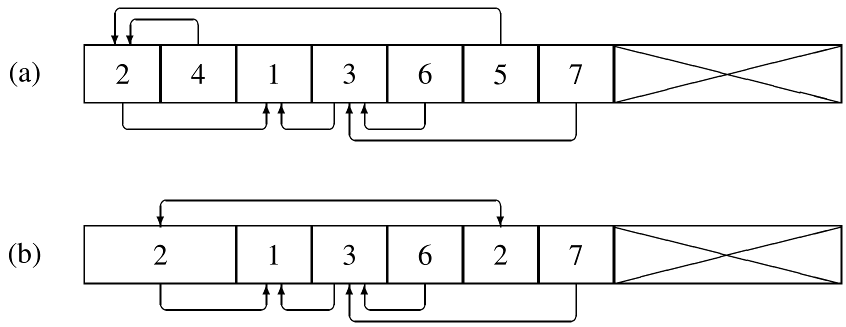


Figure 4.5: Storage layout with inter-task pointers.

We would like to use the same copying collector for interior tasks, which after all become a leaf task when resuming execution because all offspring has already ended their execution. Upon termination a child task links it private heap containing the result to the parent's heap. As a consequence the parent is no longer a single sequential block, but consists of a number of blocks scattered throughout the shared memory. The traditional sequential copying garbage collector can not be used to collect scattered such heaps since it is impossible to distinguish pointers to objects in from-space and pointers to global (ancestor) data with a single compare instruction. For example, suppose the fork-join tree of Figure 4.4a has been laid out in memory as shown in Figure 4.5a. After leaf tasks 4 and 5 have terminated and linked their heap to the parent task, task 2 resumes execution and the storage configuration changes to 4.5(b); the heap of task 2 is no longer contiguous.

When task 2 runs out of free space, it allocates a to-space at the right of task 7 and starts evacuating the live nodes. The search for pointers to live nodes in the heap of task 2 is complicated by the presence of heap 1, which breaks the simple memory layout of Figure 4.3 where global data and the from-space each have a contiguous address space. Note that task 2's internal pointers from the right part to the left part or vice versa must be distinguished from the inter-task pointers to 1. In principle the problem of distinguishing global and local data can be solved by means of a lookup table that records the owner of each storage block, but this would degrade performance because of extra memory references and table management overhead. Instead we will use a virtual address space to allocate storage such that task heaps never interleave with ancestor heaps.

The Basic Allocation Scheme, BAS

To support efficient evacuation of live data in scattered parent heaps, it is sufficient to enforce that a task's private heap is allocated to the right of all its *ancestor* heaps. This causes a strict separation of the task's (scattered) private data and its global ancestor data, so pointers can be classified with one instruction as in the sequential case. The basic allocation scheme (BAS) accomplishes the strict separation by always allocating a new heap at the right of the most recently allocated one. Virtual memory hardware is used to relocate the released physical space of the from-space to the right end after a garbage collect.

The basic scheme results in a window of physical memory moving from left to right through the virtual address space. The example in Figure 4.6 illustrates the scheme. When task 2 resumes execution in 4.6(d), its scattered heap encloses the heap of task 3, but this has no effect on the garbage collector since task 2 to does not refer to data of task 3; it only refers to data of task 1.

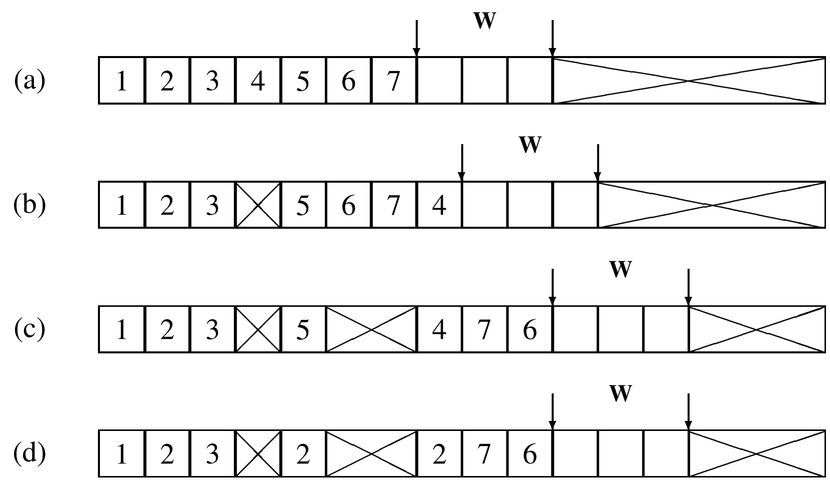


Figure 4.6: BAS: (a) initial configuration, (b) after collecting 4, (c) after collecting 7 and 6, (d) after resuming 2.

The window with available physical memory (**W**) has to be at least as large as the size of the largest private heap since tasks allocate their to-space in the window when collecting garbage. By limiting the maximum task size, we significantly lower the 50% waste of memory reserved for to-space of the (sequential) copying collectors since tasks can time-share **W** as a common to-space. The costs of this limit are that large tasks have to collect their garbage more often. Note that we can control this space-time trade-off by adjusting the value of the maximum task size. It suffices to reserve a $1/(p+1)$ fraction of the total memory size on a multiprocessor with p processors, so p large tasks can execute in parallel. If the shared to-space is a bottleneck, which we do not expect in (small) shared memory systems, a pool of to-spaces can be provided.

When the window **W** has completely moved to the right and all virtual address space has been consumed, a global action is required to reclaim the unused holes in the virtual address space that have resulted from the local garbage collects. To preserve the ordering between the tasks, the virtual space is compacted by sliding the private heaps to the left. Besides adjusting the page tables, all physical pages have to be scanned for pointers to objects in virtually "moved" pages, so they can be relocated to their new positions. This expensive compaction method limits the usefulness of the storage allocation scheme to systems where the virtual address space greatly exceeds the size of the physical memory because then compactions are rarely needed.

The Virtual Allocation Scheme, VAS

We can improve the basic memory management's rapid consumption of the virtual address space by reusing holes on the fly. Holes in the virtual address space can be freely reused for new private heaps as long as the task ordering is preserved: tasks must be allocated to the right of their ancestors. Thus, instead of always allocating memory at the right end, the Virtual Allocation Scheme (VAS) allocates a task's heap in the lowest free part of the virtual address space that lies to the right of the task's parent.

VAS works well for the common case of a divide-and-conquer application that unfolds into a task tree with small interior tasks and big leaf tasks. After the interior control tasks have divided the work into independent components, the leaf tasks run for a long time to compute the partial solutions. Under the basic storage allocation scheme these leaf tasks move to the right each time the garbage collector is invoked, but under VAS these tasks remain in a small part of the virtual address space. A leaf task that needs to allocate a to-space can usually reuse the most recently released from-space of another task since there are no allocation constraints between leaf tasks; the only constraints are between interior tasks and leaf tasks.

Figure 4.7 shows the effects of VAS for the same example as with the basic scheme in Figure 4.6. Now the positions of leaf tasks 4, 5, 6, and 7 just

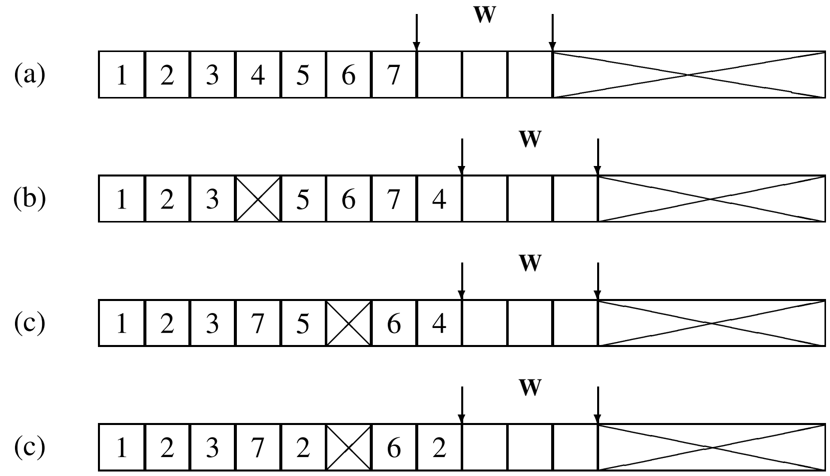


Figure 4.7: VAS: (a) initial configuration, (b) after collecting 4, (c) after collecting 7 and 6, (d) after resuming 2.

permute, but do not shift to the right. In comparison with the basic scheme, the VAS administration is slightly more complicated since it has to record the holes in the virtual address space and the position of each task's parent.

The Circular Allocation Scheme, CAS

Both previous storage allocation schemes use paging hardware to implement a large virtual address space. Obviously, this limits their applicability to multiprocessors with such hardware support, while those schemes also need a considerable amount of memory to store the page table. For example, the complete page table for a 4 Gbyte virtual address space on a MC88000 architecture with 4Kbyte pages occupies 4 Mbytes of physical memory. In addition the usage of a page as the unit of storage results in wasted heap space due to internal memory fragmentation. This has a strong effect on parallel applications that unfold into a large task tree where each interior task occupies a private page of memory that is only partially filled with useful data. Both sources of memory loss are tackled by the following allocation scheme that allocates storage in a virtual address space, but does not require paging hardware at all.

The Circular Allocation Scheme (CAS) uses a fixed translation scheme to map virtual addresses onto physical addresses. The upper bits of a virtual

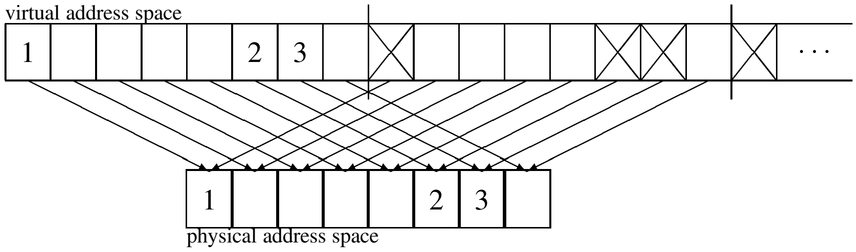


Figure 4.8: Circular Address Space

address are simply replaced by zeros to obtain the physical address. This gives a virtual address space that is wrapped circularly through the physical address space, see Figure 4.8. The "ghost" images of tasks 1, 2, and 3 cause a repeated pattern of holes in the virtual address space that extends right of the physical space.

The CAS strategy uses the same allocation policy as VAS: a task's heap is allocated at the lowest available virtual address above the task's parent. Unlike VAS, however, CAS has to skip over the *ghost* images when looking for a free hole. For example, if task 3 wants to extend its heap with another two contiguous pages to the right of task 2, then CAS cannot allocate it directly after its own heap, but has to allocate it in the large hole after the ghost image of task 1 as depicted in Figure 4.9.

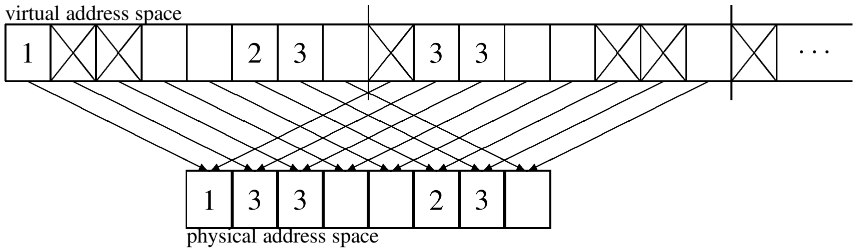


Figure 4.9: CAS after extension of task 3.

Observe that the holes in the virtual address space are just a repetition of the physical holes. To take advantage of this redundancy by recording the status of the physical space only, the CAS strategy regards virtual addresses as the concatenation of a cycle-counter (most significant bits) and a base address in the physical space (least significant bits): addr $\equiv$ cycle:base. When allocating storage to the right of a parent task located at address *cycle:base*, CAS first tries to locate a suitable hole at the right of the *base* in physical memory. If CAS succeeds then it returns *cycle:hole* as the start address of the new storage block, else CAS increases the *cycle* counter and starts looking at the beginning of the physical memory and returns $(cycle{+}1)$*:hole* on success.

If the CAS strategy fails to allocate a large contiguous block due to external memory fragmentation, the scattered free space has to be compacted by sliding the tasks down to the left. This compaction only adjusts the *base* parts of pointers, but it is more expensive than with the two previous schemes since all data has to be copied as well. In the previous example compaction is needed when task 3 in Figure 4.9 wants to allocate 3 pages to perform garbage collection. The compacted memory layout is shown in Figure 4.10.

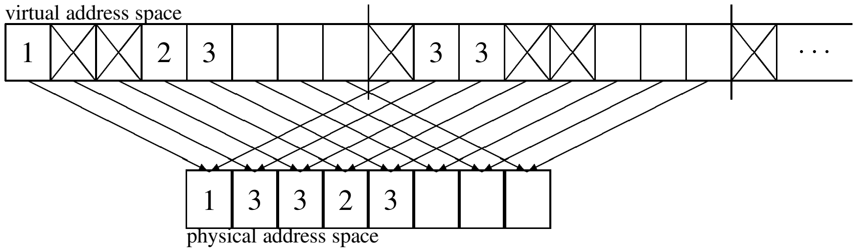


Figure 4.10: CAS after sliding compaction

Note that the sliding compaction has not compressed the virtual space, so an even more complex compaction method is needed when CAS runs out of the virtual address space: all *cycle* parts of pointers have to be cleared which requires a permutation of the tasks in physical memory to preserve the task ordering in the virtual address space.

The advantages of the CAS strategy are that there is no need to maintain the page tables since the address mapping is fixed; in fact it can be implemented in hardware by cutting the upper address pins of the processor! The fixed mapping also implies that CAS is not bound to the usage of pages, so heaps can have arbitrary sizes to avoid (internal) memory fragmentation. The CAS strategy, however, can only compete with the VAS strategy if both physical and virtual compaction operations are rarely needed.

4.4.2 Evaluation

To evaluate the performance of the three above mentioned memory allocation strategies, we have studied their behaviour by running a set of benchmark programs on the MiG multiprocessor simulator. In particular we are interested in the amount of memory wasted due to memory fragmentation, and the usage of the virtual address space. For CAS the number of physical compactions

program	runtime	# tasks	mem. usage	# garb.coll.
QUEENS	1.4	165	325,121	172
FFT	1.5	15	1,846,985	36
15-PUZZLE	28.0	24,625	28,045,720	24,736
COMP-LAB	1.6	465	1,178,347	476
WAVE	1.7	41	197,072	61

Table 4.9: Benchmark programs; the simulated runtimes in seconds are for BAS on a 4 processor system; the memory usage is the number of words (32 bits) claimed in the heap.

is also important information. Table 4.9 lists some characteristics of the five benchmark programs.

Three different versions of the runtime support system have been constructed, implementing the BAS, VAS and CAS storage allocation schemes. When tasks run out of heap space, they double their heap size if enough global memory is available, otherwise the garbage collector is invoked. When a task finishes, its result is compressed by invoking the garbage collector, after which the unused heap space is returned to the global pool. In this pilot implementation we have not directly made use of virtual memory hardware, but rather simulated the allocation schemes with one large chunk of physical memory. This suffices to collect the statistics about the memory consumption of the benchmark programs.

BAS

At first, we study the behaviour of the basic allocation scheme of Section 4.4.1, which always allocates new storage at the right end of the virtual memory space. Table 4.10 summarises the results of the benchmark programs for the basic scheme with 1024 word (= 4Kbyte) pages. The column labeled "physical" lists the maximal amount of heap words in use at any moment in time during the execution of the application. This number does not include code and static data that are located in separate segments, nor does it include the space needed for the page tables, but it does account for the memory fragmentation inside pages. The second column contains the highest virtual address used by the application, and it shows that the simplistic basic scheme consumes large quantities of virtual memory space. The 15-PUZZLE, for example, allocates 200 times as much virtual space as physically needed.

program	physical	virtual	claim rate
QUEENS	77,824	1,443,839	1.04 Mw/s
FFT	1,261,568	3,917,823	2.67 Mw/s
15-PUZZLE	726,016	142,187,519	5.13 Mw/s
COMP-LAB	208,896	4,738,047	2.89 Mw/s
WAVE	49,152	804,863	0.47 Mw/s

Table 4.10: Memory allocation statistics of the BAS strategy.

The ratio between virtual and physical memory usage depends strongly on the application's input parameters and cannot be used as a meaningful characteristic in general. Instead we have listed the application's claim rate (in Mwords/second) that shows how fast virtual memory is consumed. The high claim rate of the 15-PUZZLE is partly caused by the large number of tasks, which results in considerable memory fragmentation inside pages. The claim rate indicates how frequently a compaction of the virtual address space is needed. In our benchmark, the claim rates are limited to maximal ca. 5 Mwords/second, so an application can execute in a 1 Gword virtual address space for at least 200 seconds without a compaction on a system with four 20 MIPS processors. A 16 node processor system will (if the program has enough parallelism) consume the same virtual space in roughly 50 seconds. A compaction would take approximately 1 second per Mbyte of physical memory.

VAS

The results of using the VAS strategy are shown in Table 4.11. In comparison with the basic scheme, the benchmark applications under VAS use slightly more physical memory, but the virtual memory consumption has been significantly reduced to within a factor 2 of the application's physical memory requirement. Therefore an application is unlikely to need an expensive compaction to compress the virtual memory space, hence, the compaction operation probably does not have to be implemented at all.

The simulator records the allocation overheads, like managing the list of free pages, of the memory management schemes. The differences, however, are marginal and only account for ca. 0.5% of the total execution time in the usual case that no compactions are needed. Since this is below the accuracy of the MiG simulator we can not draw any sensible conclusions out of this number.

program	physical	virtual
QUEENS	80,896	105,471
FFT	1,261,568	1,572,863
15-PUZZLE	849,920	898,047
COMP-LAB	234,496	517,119
WAVE	49,152	73,727

Table 4.11: Performance statistics of the VAS strategy.

CAS

First we have run the benchmark programs under CAS with the same pagesize (1024 words) as the basic and VAS strategies. The results in Table 4.12 show the number of compactions to recover from physical memory fragmentation besides the physical and virtual memory usage

program	physical	virtual	compacts
QUEENS	76,800	159,743	14
FFT	1,261,568	1,703,935	0
15-PUZZLE	775,168	803,839	0
COMP-LAB	241,664	492,543	4
WAVE	49,152	73,727	0

Table 4.12: CAS performance, pagesize 1024 words.

The difference in physical memory usage under CAS in comparison to VAS is caused by their difference in allocation time, which results in different task scheduling decisions. The scheduler has a rather large influence on the amount of physical memory in use since it decides about the shape of the expanded task tree in core (breadth first vs. depth first). The virtual memory usage under CAS exceeds the physical memory usage only by a small factor, just like for VAS. Note that only the QUEENS and COMP-LAB applications perform compactions to compress the physical memory space.

Next we ran run CAS with a small pagesize of 32 words to lower the internal memory fragmentation The results are depicted in Table 4.13. Some programs need more physical and virtual memory; only the QUEENS and 15-PUZZLE benefit from the small pagesize. The increase is caused by the internal overhead to administrate the linked list of heap blocks. The “wasted” space forces the large tasks to allocate another block just before finishing their computation,

program	physical	virtual	compacts
QUEENS	49,760	109,759	9
FFT	1,586,176	2,359,295	0
15-PUZZLE	467,072	1,066,655	1
COMP-LAB	245,792	452,671	4
WAVE	60,000	134,143	10

Table 4.13: CAS performance, pagesize 32 words.

and since tasks double their heapsize when running out of storage only a small fraction is actually used.

The number of compactions listed in the performance results is a worst case value since the applications have been simulated on a multiprocessor with the minimum amount of physical memory needed by the specific application. Adding about 50% extra memory decreases the number of compactions to zero in all cases. Thus the CAS scheme performs well if the amount of physical memory in the shared-memory multiprocessor is somewhat larger than the absolute minimum required by the application.

Stressing the allocation schemes

The benchmark results for VAS and CAS show that the applications can be efficiently executed in a surprisingly small virtual address space. This is a consequence of the scheduler traversing the fork-join tree in a depth-first manner, hence at any moment the allocation strategies only have to satisfy a logarithmic number of the task allocation constraints (depth of the tree). To test the limits of the allocation schemes we therefore created a synthetic application, called *spine*, that unfolds into a degenerated tree: a linear list. The spine of interior tasks forces the allocation schemes to allocate new tasks at the right end. The results for a spine of length 512 on a 4 processor system with 1024 word pages are presented in Table 4.14.

The synthetic spine program allocates virtual address space somewhat faster than the benchmark applications: a claim rate of 8.6 Mwords/second versus 5.1 for the 15-PUZZLE. The large difference in virtual address consumption between the basic scheme and VAS is caused by leaf tasks that have allocated address space far beyond the growing spine: whenever such a leaf task finishes its computation, the garbage collector is invoked to compress the result and the reclaimed space at the right of the spine can be reused for new tasks. The

strategy	physical	virtual	comp	claim rate
BAS	393,216	17,105,919	-	8.6 Mw/s
VAS	393,216	2,117,631	-	1.1 Mw/s
CAS	393,216	793,599	27	0.4 Mw/s

Table 4.14: Performance statistics of *spine*.

CAS strategy needs even less virtual address space because of the 27 physical compactions: they also reclaim the virtual address space that resides in the currently highest cycle. The need for compactions, however, is probably a disadvantage because of the additional performance costs for CAS.

The total amount of virtual space claimed by the *spine* program can be made arbitrarily large by increasing the length of the spine, but the moderate claim rate limits the virtual compaction frequency to a low value for all three memory management strategies.

4.5 Discussion

The WYBERT design for graph reduction on shared memory multiprocessors differs considerably from the other parallel implementations of functional languages on shared memory multiprocessors: $<\nu,G>$, AMPGR, and GAML. First, WYBERT only supports the high-level divide-and-conquer skeleton (the sandwich annotation) for generating parallel tasks instead of the general spark-and-wait model. This limits the class of applications, but many problems fit the divide-and-conquer paradigm directly or can be mechanically transformed. Second, the complication of shared redexes in the global address space is avoided by the FRATS reduction strategy, which eagerly evaluates shared data before sparking parallel tasks so shared redexes do not exist. As a consequence there is no need to lock application nodes in the heap to enforce consistency in face of updates as in the general case of parallel graph reduction on shared memory multiprocessors.

A potential disadvantage of FRATS is that it evaluates unneeded expressions, but the discussion in Section 4.2 has shown that the problem of superfluous computation can be handled by applying a few program transformations. These transformations were demonstrated to be both successful and necessary: three out of six benchmark programs did not terminate when executed under FRATS, whereas all transformed programs ran to completion without any significant overhead. At the moment these transformations have to be applied by hand, but the application of the important cycle-naming transformation can easily be automated: the compiler can find the super set of (recursive) invocations of curried functions with unevaluated parameters without any difficulty.

The divide-and-conquer parallelism combined with the FRATS reduction strategy results in a tree-shaped task graph with independently executing coarse-grain leaf tasks. Remember that the grain size has to be controlled explicitly by the programmer. WYBERT takes advantage of the runtime application behaviour in its storage management policy:

- Garbage collection is performed per leaf task individually instead of having all processors synchronise when running out of free space. Besides omitting the need for low-level synchronisation during garbage collection, the to-space of the copying collector can be time-shared between multiple processors to reduce the fraction of wasted space: $1/(p+1)$ instead of $1/2$ when memory is equally divided among p processors. Note that the maximum task size is limited by the selected size of the to-space.

  The usage of a sequential copying collector for resumed tasks requires a storage manager that allocates the scattered heap of a task to never interleave with an ancestor heap. The benchmark results in Section 4.4 for three such storage management schemes show that this can be efficiently accomplished with virtual memory hardware provided that the size of the virtual address space is three times as large as the amount of physical memory in the multiprocessor.

- The stack for graph reduction is not allocated one per task, but one per processor. All tasks executing on a processor share the same stack, and fork tasks leave their context on the processor stack until all children have terminated. This provides efficient stack-based graph reduction and fast context switching, but constrains the scheduler since only the top-most task on the processor stack may execute.

  The WYBERT scheduler is not hampered by the ToS constraint as shown in Section 4.3. The worst case of loosing all parallelism is never observed for the benchmark programs: ToS performs on average within 2% of the general list scheduling policies. To effectively use the caches, the ToS scheduler employs local tasks pools per processor managed in LIFO order. This causes a depth first traversal of independent subtrees, scheduling the lastly created task whose context still resides in the cache first. In addition, the depth first order minimises the resource usage and number of active tasks as well. When running out of work, processors steal (large) tasks in FIFO order to minimise synchronisation overhead.

Thus each task is provided with a private heap and executes on top of the shared processor stack in contrast to general spark-and-wait systems where tasks claim nodes in the shared heap and own a private stack.

Based on the studies presented in this chapter we conclude that WYBERT is a feasible design. To asses the absolute performance of WYBERT in comparison to other implementations we have constructed a prototype implementation, which will be discussed in the following chapters. Note that it is impossible to give an accurate prediction of the performance by means of a high level simulation model since the fundamental advantage of avoiding the locking of graph nodes requires a study at the memory-access level.

Chapter 5

The FAST/FCG compiler[†]

To test out the WYBERT design for parallel graph reduction on shared memory multiprocessors in practice, considerable effort has been put in the development of a prototype implementation. The goal of achieving high absolute performance, not merely perfect speed-ups, requires the usage of a state-of-the-art compiler. This chapter describes the code generator of the FAST/FCG compiler, which has been designed to meet the requirements of compiled graph reduction in general and WYBERT in particular. For example, the code generator is targeted towards a copying garbage collector and tags all data values such that pointers and basic values can be quickly distinguished. To keep the FAST/FCG compiler as general as possible, most WYBERT specific code has been hidden in the runtime support system; only the sandwich annotation is handled as a special case: the compiler generates code to squeeze task arguments and then call a RTS function to spark the tasks for parallel execution. This approach is possible because of the clear separation between graph reduction (compiler) and parallelism (RTS) in the WYBERT design.

In comparison to ordinary imperative compilers, a functional language compiler has to take care of all the extra expressiveness that such a language offers. A popular method for compiling functional languages is to make maximal use of existing imperative compiler technology by constructing a front end that translates a functional program into imperative code (e.g., C) [Schulte91, Peyton Jones92]. Higher order functions and lazy evaluation are typically handled by support functions that manipulate heap allocated closures holding a function identifier and some arguments. The quality of the generated code heavily depends on the front end's strictness analysis that minimises the

[†]This chapter represents joint work with Pieter Hartel.

inefficient usage of these closures. Although the "generate-C" method requires minimal implementation effort, the C-compiler hampers performance of the resulting object code because it prohibits control over pointers that are stored in the processor's registers and stack. This control is crucial for two reasons:

- Efficient garbage collection algorithms like two-space copying and generation scavenging require all pointers into the heap to be known to the garbage collector because objects are moved and pointers have to be adjusted accordingly.
- Frequently accessed pointers like the pointer to the start of free heap space should be stored in global registers.

For maximal performance the code generator needs intimate knowledge of the location of pointers on the calling stack and in registers. Therefore several functional language compilers have adopted the do-it-yourself method of generating assembly code directly [Johnsson84, Loogen89, Smetsers91]. This alternative approach gives total control over all pointers and the processor, but it goes against the grain of the *lazy* implementor because now we have to deal with important low-level issues like register allocation and code scheduling, while this can be done perfectly well, and probably better, by (part of) an existing C compiler. Besides implying extra work, this "generate-assembly" method looses on portability as well, since C compilers are available for almost any type of computer.

Our compiler combines the advantages of both previous approaches: it compiles down to a level where it has control over the location of pointers and then uses part of the C compiler to generate object code. We have made use of the existing FAST front end [Hartel91a], which includes an advanced strictness analyser. The front end translates a functional program into a severely restricted subset of C, which is called Functional C, with standard call-by-value semantics. Since pointers are passed as ordinary parameters, direct compilation of Functional C results in code that cannot be used in combination with moving garbage collectors. Therefore the Functional C Code Generator (FCG) compiles the FAST output further to code (KOALA) that uses an explicit call stack, which brings all pointers under control of the garbage collector.

source program → $\boxed{\text{FAST}}$ → *Functional-C* → $\boxed{\text{FCG}}$ → *KOALA* → $\boxed{\text{translator}}$ → *C* → $\boxed{\text{gcc}}$ → *object code*

Figure 5.1: Compiler structure.

The complete compiler is organised as a pipe-line of four programs:

1 The Fast front end translates functional programs to Functional C, thereby making the functional expressiveness explicit by inserting calls to support functions as with the "generate-C" approach.
2 The FCG code generator compiles the FAST output to the KOALA assembly frame work that supports features like register allocation and code scheduling.
3 The third phase translates the KOALA code into low-level C.
4 Finally the GNU gcc compiler is used to generate the actual object code.

The reason for translating KOALA into C is twofold: portability and reuse of existing compiler technology. A disadvantage of this approach is the poor compilation speed. If this becomes a problem, a future release of the translator should directly interface with the intermediate code level of the GNU gcc compiler.

Before discussing the FCG code generator in detail in Section 5.4, a short description of the FAST front end, KOALA assembler, and graph representation is given to introduce the source and target language of the code generator. The chapter concludes with a comparison of FAST/FCG and other competitive compilers for lazy functional programming languages.

5.1 The front end

The FAST compiler, which has been developed at Southampton University, forms the first stage in our compiler pipe-line. It accepts programs written in a small lazy functional language called *Intermediate*. Intermediate is similar to the functional language Miranda, providing higher order functions, lazy evaluation, and pattern matching in function definitions. Intermediate does not support Miranda's operator overloading, arbitrary precision numerals, "offside rule", and module system.

Function definitions in Intermediate have the form of a set of recursive equations with the possibility to express list pattern matching on function arguments. `[]` represents the empty list and the cons operator is denoted by the colon (`:`). The language is higher order, curried, and lazy.

The function *append* in Figure 5.2 gives an example of an Intermediate program. It uses list pattern matching on its first argument to decide whether to recurse on the tail of the list, or to return the second argument.

The FAST compiler produces as output equivalent Functional C programs with call by value semantics. Functional C serves as the source language

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

Figure 5.2: Append in Intermediate.

for the back end and is essentially a subset of C, see the syntax given in Figure 5.3. The main restrictions are directly related to the functional style of programming: single assignment of local variables, no global variables, and if-then-else as the only control structure. Functional C supports only one type, namely **ptr**, which may be a basic value or a pointer into the heap, and relies on the primitive functions to correctly interpret their operands. For example, *add_i* operates on integers, while *and_b* uses booleans. As a consequence all types in Functional C must have the same size; therefore data structures are represented as a pointer to a sequence of fields in the heap.

The Functional C code for *append* as shown in Figure 5.4 has essentially the same structure as the program in Figure 5.2. There are three important differences: firstly the implicit laziness of the Intermediate version is now explicit in the form of the calls to the library functions *reduce* and *vap* (for vector application). The latter builds a suspension of a function (*append* in this case) in the heap, and the former evaluates a previously built suspension. Thus all functions present in Figure 5.4 can safely and efficiently be called with call by value semantics, as is the usual case in C programs. The second difference between figures 5.2 and 5.4 is, that pattern matching and other list operations

program	::=	function$_1$ $\cdots$ function$_p$	
function	::=	**ptr** id(id$_1$**,** $\cdots$ **,** id$_f$) decl **{**decl body**}**	
decl	::=	**ptr** id$_1$**,** $\cdots$ **,** id$_d$**;**	
body	::=	assignment **;** body	
	|	**return** expr **;**	
	|	**if** (expr) **{**body$_t$**}** **else** **{**body$_e$**}**	
assignment	::=	id **=** expr	
expr	::=	id(expr$_1$**,** $\cdots$**,** expr$_e$)	
	|	id**[**num**]**	/* array subscription */
	|	id	

Figure 5.3: Functional C.

```
ptr append( x_xs, ys)
ptr x_xs, ys;
{
  ptr x, xs;

  if( null(x_xs)) {
    return reduce(ys);
  }
  else {
    x  = x_xs[0];     /* head */
    xs = x_xs[1];     /* tail */
    return cons(x,vap(prel_append,xs,ys));
  }
}

ptr prel_append( vap)
ptr vap;
{
  return append(reduce(vap[1]),vap[2]);
}
```

Figure 5.4: Append in Functional C.

are now compiled into calls to library routines, such as *head*, *cons*, and *null* for taking the head of a list, constructing a list, and testing for an empty list respectively.

The third difference, which is not apparent from the example, is that statements in Functional C are explicitly ordered, which is indicated by the sequencing '**;**', while expressions in a lazy functional program are implicitly ordered by their dependencies. The dependencies that have been discovered by the compiler do not require interpretation at runtime.

The presence of calls to *reduce* requires runtime interpretation of the graph that resides in the heap; the argument of *reduce* points to a closure in the heap that has to be evaluated to head normal form. This is a much less efficient way of evaluating an expression than obeying straight sections of C code. The FAST compiler makes strenuous attempts to avoid interpreting the graph whenever possible, so there are far fewer occurrences of *reduce* than a naive front end would generate. This streamlines the underlying evaluation mechanism, and the compiler employs a host of other analyses to further improve the quality of the generated code.

The FAST compiler generates Functional C code based on the principle that the callee decides what form its arguments should have. Some functions require arguments that are evaluated, while other arguments may be passed as is. The null test for example is strict in its argument, because *null* needs to inspect the argument to see whether or not it represents the end of a list. The function *cons* on the other hand does not need to know anything about its arguments as it merely combines them into a new data object. Based on information about primitive functions, the strictness analysis phase of the compiler works out that *append* is strict in *x_xs* but not strict in *ys*.

A second principle, which is inherent to lazy evaluation, is that every function when it is actually called (as opposed to being embedded in a suspension), will return an evaluated object. *Append* must therefore call *reduce* explicitly to guarantee that when returning *ys*, this parameter has actually been evaluated to head normal form.

The principle that the callee decides on the form of its arguments has an interesting consequence for the organisation of the generated code. Returning to Figure 5.4, we see that functions are not embedded directly in a suspension, but via another "prelude" function, which in the case of *append* is *prel_append*. When a suspension is evaluated, the prelude function first calls reduce for every strict argument before calling the function proper. *Prel_append* ensures that the strict first argument is indeed evaluated before entering *append*. The non-strict second argument is merely passed on.

As an optimisation the compiler generates prelude functions specialised towards each call site, so when it is actually known at compile time that a particular argument has the required form, no redundant calls to reduce are made. A function may be partially applied, in which case the prelude function assumes that the as yet missing arguments will eventually be supplied in unevaluated form.

5.2 The assembler

The KOALA assembler forms the last stage in the compiler pipe-line. It serves as the target for the FCG code generator, and is presented first to give a better understanding of the optimisations employed in the code generator as described in Section 5.4. KOALA is a high-level "assembly" frame work that provides a simple abstract machine suitable for graph reduction. It resembles the G-machine [Johnsson84], and consists of a CPU, an *unlimited* number of registers, a stack, and memory. KOALA's instructions are listed in Figure 5.5.

fetch *n reg*	fetch the *n*-th stacked value into register *reg* $(n \geq 1)$.
push *id*	push *id* on the stack; *id* refers to a register or constant.
pop *reg*	pop the value on top of the stack into register *reg*.
dup *n*	duplicate the *n*-th stacked value on top of the stack.
squeeze *n m*	slide down the top *n* elements of the stack, squeezing out the *m* below.
store reg_{src} reg_{addr}	store the contents of reg_{src} in memory at the location specified in reg_{addr}.
load reg_{addr} reg_{dest}	load one word at location reg_{addr} from memory into register reg_{dest}.
label *lbl*	instruction label (pseudo instruction).
branch id_{dst}	unconditional branch; id_{dst} is (the contents of) a register or a label.
bfalse *lbl*	pop boolean from the stack and branch to *lbl* if it is false.
jfalse *reg lbl*	jump if the boolean value in register *reg* is false.
fun *name*	function entry point (pseudo instruction).
call *fun*	branch to function *fun*.
return	return to caller; pop return address from the stack.
move id_{src} *reg*	move id_{src} to register *reg*; id_{src} refers to a register or constant.
alu *op* $id_1 \cdots id_n$ *reg*	parameterised *n*-operand instruction: $reg = op\ id_1 \cdots id_n$ *op* is a basic alu operation: add, mul, etc. id_i is either a constant or a register.

Figure 5.5: KOALA's instruction set.

The KOALA stack is used to implement the function call mechanism: parameters are passed via the stack and the local state of a function is saved on the stack when calling a (recursive) function. The return address is passed as an extra parameter that will be used as branch destination on function return. This simple calling sequence does not include a frame pointer, so each function has to **squeeze** the call stack into a proper state before returning to its caller. If the result value is returned via a register then this amounts to just popping some items off the stack; this is cheaper than maintaining a frame pointer, which has to be saved and restored on function calls. To support easy integration into a parallel implementation, we have restricted KOALA to one single stack that combines the multiple stacks found in other abstract reduction machines (see Section 3.1.3).

For simplicity the heap can only be accessed via basic **load** and **store** instructions that transfer one word between a register and memory. In particular there are no high level instructions to allocate heap cells because the FCG code generator will perform certain optimisations on heap bound checks like inlining and clustering that would make those tests redundant inside the allocate instruction.

The minimal set of control instructions provides enough functionality to implement the function call/return sequence and the if-then-else construct present in Functional C. The remaining instructions that do the actual arithmetic computations, logical operations, etc. are provided by the **alu** instruction, which takes the specific operation as it first argument.

The description of the KOALA instruction set contains no notion of data types; every item is regarded as some value that fits into a word of the underlying machine, which corresponds both to Functional C's uniform usage of the **ptr** type, and to a simple bit pattern at machine level. The ALU functions decide how to interpret the bit pattern, as an integer, boolean, floating point number, etc. The unlimited number of (virtual) registers makes KOALA different from traditional assemblers.

5.2.1 Implementation

All KOALA instructions that manipulate the stack, like **push** and **pop**, can be expanded straightforwardly into a few kernel instructions (**alu** + **load/store**), which operate with a fixed top-of-stack register. The subset of KOALA that then remains, matches with the traditional intermediate code of imperative compilers like the three-address code described in [Aho86]. Unfortunately not many compilers are capable of reading-in some intermediate code file, let alone that there exists a universally agreed upon format. Therefore we have taken the detour of translating KOALA back to C, which will then (again) be translated into some intermediate code by the C compiler itself.

A nuisance with using C as a sophisticated assembly language is that standard C does not support code labels properly: it forbids the usage of labels in expressions. This makes it impossible to directly push a (C) label as return address on the stack (i.e. store it in memory). The work-around is to use one level of indirection: KOALA labels are encoded as integers, and branches are translated to indirect gotos:

<table>
<tr><th>KOALA</th><th>C</th></tr>
<tr><td>label lbl

branch lbl
branch reg</td><td><pre>lbl:

goto lbl;
dest = reg; goto jump;

jump: switch (dest) {
        case 0: goto lbl_0;
        case 1: goto lbl_1;
                   ⋮
      }</pre></td></tr>
</table>

Note that only branches with a register target suffer this indirection. Examination of SPARC assembly code showed that such an indirect branch expands into 9 machine instructions. Hand patching of indirect to direct jumps in the assembly code of the function-call intensive nfib benchmark program, reduces the runtime to 70%. For "real" programs that contain large basic blocks, however, the difference will be considerably less. Functional programs tend to have longer basic blocks than imperative programs because of the lazy semantics that cause the construction of complicated graph structures that represent local definitions in (large) where clauses.

Our KOALA-in-C implementation translates a complete KOALA program into one single C function. This stresses most C compilers since they usually generate code for a procedure at once, but in return our method produces "globally" optimised code. It is, of course, possible to disassemble KOALA into C style functions, but this would introduce inefficiencies like maintaining an explicit pointer stack for the garbage collector. Experience with the SUN and GNU C-compilers has shown that the SUN compiler with optimisations enabled gives up on large programs due to swapping problems, while the GNU compiler on the contrary executes faster with optimisations asserted than without.

The generated object code for the SPARC matches well with the KOALA source; in particular the GNU C compiler manages to assign KOALA's virtual registers to the sparc processor's physical registers without spilling values to the C stack.

5.3 Graph representation

Both the graph reducer and the garbage collector operate on (pointers to) objects allocated in the heap. Their efficiency depends on the encoding of pointers and objects, hence, it is important that the data representation scheme supports the different requirements:

garbage collection: The storage management of WYBERT is based on a two-space copying garbage collector [Cheney70]. The collector scans the call stack for root pointers to live objects in the heap, hence, the collector has to be able to distinguish between pointers (to objects in the heap) and other data types: basic data values like integers, return addresses, etc. Since the collector copies all live objects into the empty semi-space to compact the graph, the collector has to know the size of each object and the location of all pointer fields within an object.

graph reduction: The reducer allocates heap objects not only to hold data structures like lists, but also to hold suspended computations. When accessing components of heap allocated objects, the reducer must assure that the pointer refers to an evaluated object, not to a suspension. Therefore the reducer has to check for a delayed computation at runtime whenever it dereferences a pointer in a non-strict context (for the first time).

The graph reducer assumes that programs are correctly typed, so the reducer never has to determine the type of an object. This slight restriction rules out untyped languages such as LISP and SASL, but saves a large amount of runtime checks and boosts performance [Appel89].

The requirement of the garbage collector to be able to distinguish pointers from other data is rather a nuisance since either all data has to be tagged or the compiler has to generate information about the layout of each and every object type and stack frame. The latter solution has the advantage that no tag-handling is required, but it complicates matters considerably. For example, the spineless tagless G-machine [Peyton Jones92] incorporates a special pointer stack and generates garbage collection information for each object type. For convenience we have adopted the tag-it-all solution and the performance results in Section 5.5 show that the tag handling overhead can be kept small.

The pointer/data classification of the garbage collector does not match well with the reducer's test whether a heap pointer refers to a suspension or a data object; pointers should be further classified into two categories: constructor and application pointers. Thus in total three data types should be efficiently recognizable. To avoid wasting a whole word of memory for a two-bit tag value, the tag is encoded in the least significant bits of the pointer or data. In case of pointers, the tag bits come for "free" since heap objects have to be aligned on four byte boundaries in memory anyway. Another advantage is that

<table>
<tr><th>Pointer</th><th>Type</th><th>Description</th></tr>
<tr><td>xxxx1</td><td>Basic</td><td>Basic data types like integers and floating point numbers are encoded in the pointer itself; xxxx represents the 31-bit value.</td></tr>
<tr><td>xxx10</td><td>Cons</td><td>xxx00 points to a data Constructor whose first (header) field provides additional type information:
<table>
<tr><th>header</th><th>type</th><th>description</th></tr>
<tr><td>yyy00
yyy10</td><td>list</td><td>The list constructor consists of a head and a tail field. The tail is the first(!) field and points to either another list constructor (case yyy10) or a vector apply node (case yyy00).</td></tr>
<tr><td>yyy01</td><td>curried</td><td>Curried function nodes hold a function identifier (code pointer) and a number of arguments that is less than the function's arity. yyy (= cccas) contains three bit-fields that encode attributes associated with the curried node:
<table>
<tr><td>ccc</td><td>20 bits</td><td>code address</td></tr>
<tr><td>a</td><td>5 bits</td><td>function arity</td></tr>
<tr><td>s</td><td>5 bits</td><td>#arguments in the node</td></tr>
</table></td></tr>
</table></td></tr>
<tr><td>xxx00</td><td>VAP</td><td>xxx00 points to a Vector APply node; the header field is used to distinguish between two types:
<table>
<tr><th>header</th><th>type</th><th>description</th></tr>
<tr><td>yyy00
yyy10</td><td>application</td><td>A function application has two fields: the first field contains the function part that points to either another VAP node (case yyy00) or a curried function (case yyy10), while the second field holds the argument that can be of any type.</td></tr>
<tr><td>yyy01</td><td>suspension</td><td>A suspended function is encoded just like a curried function, but the number of arguments is equal to the function's arity. If the size field is 0 then the node represents a Constant Applicative Form (CAF).</td></tr>
</table></td></tr>
</table>

Figure 5.6: Data representation.

the graph reducer does not have to make a memory reference to inspect the tag as is the case for data representation schemes that encode the tag in the object.

A schematic overview of the data representation of the WYBERT prototype implementation is presented in Figure 5.6. The graph reducer can easily check if a data object is in head normal form by inspecting whether the two least significant bits of the pointer to the object are set to zero or not. The garbage collector only has to test the lowest bit to recognise a pointer.

All heap objects are encoded as a sequence of tagged words. The size of the object, which is needed by the copying garbage collector, is included in the first word of an object that serves as a header field. This header contains some additional information about application nodes for the graph reducer as well: the code address of the function's entry point, and the function's arity. The arity is needed to test whether a curried function is applied to enough arguments or not. In the latter case a new, but larger, curried function has to be returned. The size, arity, and code-address are encoded in one field for space efficiency. Furthermore these headers are encoded as basic values, i.e. the least significant bit is set to 1, so that the garbage collector will automatically skip them when scanning moved nodes for pointers into live data.

The data representation scheme has been optimised to avoid the header field for two important object types: lists and unary function applications. The first field of an application node points to either another application node (*yyy00*) or a curried function node (*yyy10*). In both cases the lower tag bits of the first field serve to distinguish the node from a vector apply node that starts with a header (*yyy01*). Likewise the tail of a list can only point to another list (*yyy10*) or application (*yyy00*), so it can serve as a header field too; the head of a list cannot be used for this purpose since it might contain a basic value, which has the same tag as the header info! As a result the list and unary function application nodes occupy two words instead of three.

To preserve the sharing of delayed computations, the graph reducer updates a vector apply node with its result. However, since the tag is encoded in the pointer to the node, the reducer cannot change the type of the node to, say, basic value. Therefore the reducer overwrites the first (header) field with the identity function and stores the result in the second field. To avoid the overhead of a function call on subsequent uses, the reducer recognises the indirection nodes especially and simply fetches the previously computed value. The garbage collector reduces the overhead even further by updating the pointers to indirection nodes with the result value during the scan of live data nodes.

A second type of indirection node has been used to support the squeeze operation of the FRATS reduction strategy. These *normalised indirection* nodes signal objects (data structures and partial function applications) that have been fully evaluated to normal form. Whenever the squeeze detects such a normal indirection it returns immediately, while in case of a plain indirection node the "value" has to be inspected for pointers to yet unevaluated suspension nodes. If two tasks share a large data structure (e.g., a matrix) the usage of normalised indirection nodes saves the (second) redundant traversal of the shared data structure. During ordinary execution the two types of indirection nodes are handled completely similar.

In comparison with data representation schemes that do not use pointer bits to encode tags, but always include a tag in an object, our method has two advantages: First, the graph reducer saves memory references since it does not have to fetch tags from memory to decide whether an argument has already been evaluated or not. Secondly, the nodes are encoded as space efficient as possible, which reduces the number of garbage collections, and improves cache locality as well. The disadvantages of our scheme are the usage of indirection nodes and the tagging/masking of basic data values, but the performance results in Section 5.5 show that the overheads are not significant.

5.4 The code generator

The back end forms the middle stage in the compiler pipe-line, and translates Functional C code into KOALA assembly. The main objectives of the FCG code generator are to allocate frequently accessed pointers into registers for efficiency and to make all heap pointers accessible during garbage collection. The former requirement amounts to allocating the start-of-free-space and end-of-heap pointers into fixed KOALA registers, while the latter is accomplished by saving all local state on the KOALA stack on function calls. Since data values are tagged, the garbage collector can easily identify all root pointers that reside on the KOALA stack.

In Figure 5.7 the following three compilation schemes are used to show how FCG implements the function-call mechanism in KOALA:

$\mathcal{K}[\![\mathit{function}]\!]$ The top-level scheme generates code for a function definition.

$\mathcal{R}[\![\mathit{body}]\!]\ \rho\ d$ The $\mathcal{R}$eturn scheme generates code to return the value produced by $body$, where d is the current depth of the stack frame, and ρ is an association list (symbol table) that maps variables to their location in the frame.

$\mathcal{K}$ :: function $\rightarrow$ [instr]
$\mathcal{R}$ :: body $\rightarrow$ [instr]
$\mathcal{E}$ :: expr $\rightarrow$ [instr]

(0) $\mathcal{K}[\![$ptr fun(id$_1$,$\cdots$,id$_n$) dcl $\{$dcl body$\}]\!]$	= **fun** fun_$n+1$;	
	$\mathcal{R}[\![$body$]\!]$ [<cont,n>,<id$_1$,n-1>,$\cdots$,<id$_n$,0>] (n+1)	
(1) $\mathcal{R}[\![$id = expr; body$]\!]$ ρ d	= $\mathcal{E}[\![$expr$]\!]$ ρ d	
	$\mathcal{R}[\![$body$]\!]$ (<id,d>:ρ) (d+1)	
(2) $\mathcal{R}[\![$return expr;$]\!]$ ρ d	= $\mathcal{E}[\![$expr$]\!]$ ρ d	
	$\mathcal{E}[\![$cont$]\!]$ ρ (d+1)	(stack return address)
	squeeze 2 d;	
	return;	
(3) $\mathcal{R}[\![$if (expr) $\{$body$_t\}$ else $\{$body$_e\}]\!]$ ρ d	= $\mathcal{E}[\![$expr$]\!]$ ρ d	
	bfalse lbl;	(fresh label)
	$\mathcal{R}[\![$body$_t]\!]$ ρ d	
	label lbl;	
	$\mathcal{R}[\![$body$_e]\!]$ ρ d	
(4) $\mathcal{E}[\![$fun(expr$_1$, $\cdots$, expr$_n$)$]\!]$ ρ d	= $\mathcal{E}[\![$expr$_n]\!]$ ρ d	
	$\vdots$	
	$\mathcal{E}[\![$expr$_1]\!]$ ρ (d+n$-$1)	
	$\mathcal{E}[\![$lbl$]\!]$ ρ (d+n);	(stack fresh label)
	call fun_$n+1$;	
	label lbl;	
(5) $\mathcal{E}[\![$id[n]$]\!]$ ρ d	= $\mathcal{E}[\![$field(id,'WORDSIZE*n')$]\!]$ ρ d	
(6a) $\mathcal{E}[\![$id$]\!]$ [$\cdots$, <id,p>, $\cdots$] d	= **dup** (d$-$p);	(variable)
(6b) $\mathcal{E}[\![$id$]\!]$ ρ d	= **push** id;	(global name)

Figure 5.7: FCG's compilation rules to KOALA instructions.

$\mathcal{E}[\![expr]\!]\ \rho\ d$ The $\mathcal{E}$xpression scheme generates code to compute the head normal form of $expr$. It puts (a pointer to) the value on top of the stack.

When calling a function the caller constructs a call-frame by evaluating the parameter expressions one by one on top of the stack and pushing a return address. Then a jump is made to the function entry point, see rule (4) in Figure 5.7. When the callee has computed the result, it fetches the return address from the stack and removes its call frame from the stack with the **squeeze** instruction, while leaving the result on top of the stack, see rule (2). The presence of the return address as an extra parameter is made explicit in rule (0) where the identifier cont(inuation) is inserted in the symbol table.

According to the syntax of Functional C (Figure 5.3), assignments to local variables only occur at the beginning of basic blocks. This restriction guarantees that whenever an assignment is encountered, the call stack contains only parameters and variables, but no anonymous temporary expressions. Rule (1), which compiles the assignment statement, therefore simply extends the call frame by calling the $\mathcal{E}$ scheme and records the location in the association list (ρ) for compilation of the remainder. This contrasts with the common technique of allocating space for all local variables at once at the function entry. Our method of not allocating complete call frames at the function entry has several advantages:

- There is no need to initialise variables on function entry to keep the garbage collector from chasing arbitrary pointers placed on the stack sometime earlier. Omitting the initialisation might (and will!) lead to a crash of garbage collectors that move objects since it is possible to find an old pointer (to a deallocated object) on the stack that now points in the middle of a new object.
- The size of the stack frames is usually smaller because variables are allocated on demand. If a function calls another function then only variables that have already been assigned are saved on the stack and no space is wasted for variables that will be assigned when control returns. Furthermore, if the then and else branch of an if-statement use a different number of variables then the actual number of variables is saved in each branch when calling a function instead of the maximum number.
- It is easier to optimise a stack without "holes".

Rule (3) in the compilation schemes handles the if-then-else control flow construct of Functional C; note that both branches use the $\mathcal{R}$eturn scheme to compute the function result, hence no additional trailing code is necessary. The array subscript is syntactic sugar for the *field* primitive, which loads a word from the heap at the location specified by the base and offset arguments, rule (5). Finally, rule (6) handles the evaluation of identifiers. It distinguishes between two types: variables, which are to be found on the call stack, and global names that refer to (constant) functions.

The $\mathcal{K}$, $\mathcal{R}$, and $\mathcal{E}$ compilation schemes generate a subset of the KOALA instruction set: only the pure stack instructions (i.e. the ones that do not have register operands) are being used. This is a consequence of using Functional C, which contains no built-in operators, but calls primitive functions instead. These primitives are directly coded in KOALA and exercise the remainder of the KOALA instructions (e.g., alu, load, and store).

The usage of the FCG compilation schemes of Figure 5.7 in combination with the KOALA-to-C translator results in poor runtime performance of the generated object code. The C compiler, which is used in the final stage in our compiler pipeline, does not "understand" the meaning of KOALA's stack instructions and faithfully compiles every **push** and **dup** instruction to loads and stores. The C compiler cannot properly optimise basic blocks by keeping temporary stacked values in registers. To make full use of the C compiler's optimisation capabilities, we therefore present some optimisation schemes that transform the FCG's stack code into a form that is amenable to optimisations by the C compiler; of great importance are those optimisations that replace stack instructions by register moves.

To illustrate the effects of the various optimisations, we will use the append function (Figure 5.4) as an example throughout the remainder of this section. A quantitative analysis of the various optimisations is provided in Section 5.5. The unoptimised compiler schemes of Figure 5.7 produce the following code for *append*:

```
fun append_3;  dup 2;
               push L0;
               call null_2;
label L0;      bfalse L1;
               dup 3;
               push L2;
               call reduce_2;
label L2;      dup 2;
               squeeze 2 3;
               return;
label L1;      push #0;
               dup 3;
               push L3;
               call field_3;
label L3;      push #4;
               dup 4;
               push L4;
               call field_3;
label L4;      dup 5;
               dup 2;
               push prel_append;
               push L5;
               call vap_4;
label L5;      dup 3;
               push L6;
               call cons_3;
label L6;      dup 4;
               squeeze 2 5;
               return;
```

5.4.1 Tail call optimisation

Both branches of the if-then-else construct in *append* end by returning the value of a function call (*reduce*, and *cons* respectively). The corresponding KOALA code evaluates the function call, reorders the stack frame, and returns the (unmodified) result. At runtime this results in *reduce*/*cons* jumping back to *append*, squeezing the call stack, and jumping back to *append*'s caller. The

$$\begin{array}{lll}
\text{(2a) } \mathcal{R}[\![\text{return fun(expr}_1, \cdots, \text{expr}_n\text{);}]\!]\ \rho\ \text{d} = & \mathcal{E}[\![\text{expr}_n]\!]\ \rho\ \text{d} & \\
& \vdots & \\
& \mathcal{E}[\![\text{expr}_1]\!]\ \rho\ \text{(d+n}-1\text{)} & \\
& \mathcal{E}[\![\text{cont}]\!]\ \rho\ \text{(d+n)} & \text{(stack return address)} \\
& \textbf{squeeze}\ (n+1)\ \text{d;} & \\
& \textbf{branch}\ \text{fun_}n+1\text{;} &
\end{array}$$

Figure 5.8: FCG compilation rule for tail calls, to be inserted before rule 2 in Figure 5.7.

sequence of jump-to-caller instructions can be collapsed into one by reordering the stack frame before the (tail) call, and passing *append*'s return-address on to the primitive call. This can easily be accomplished by extending the $\mathcal{R}$ scheme to include a special case as shown in Figure 5.8.

The effect of rule (2a) can be seen in the following code; revision bars indicate the difference with the naive code:

```
fun append_3;   dup 2;
                push L0;
                branch null_2;
label L0;       bfalse L1;
                dup 3;
                dup 2;
                squeeze 2 3;
                branch reduce_2;
label L1;       push #0;
                dup 3;
                push L3;
                call field_3;
label L3;       push #4;
                dup 4;
                push L4;
                call field_3;
label L4;       dup 5;
                dup 2;
                push prel_append;
                push L5;
                call vap_4;
label L5;       dup 3;
                dup 5;
                squeeze 3 5;
                branch cons_3;
```

5.4.2 Compile-time stack simulation

To improve the KOALA code by introducing registers we need to simulate the stack at compile time. Then it will be possible to replace matching **push** and **pop/fetch/dup** instructions by (register) **move**s inside basic blocks. Basic blocks are delimited by **fun**, **call**, **branch**, and **return** instructions.

In contrast with the tail-call optimisation, we do not enhance the basic FCG compilation schemes, but provide an $\mathcal{A}$ssembly scheme that will be used as an optimising filter on KOALA code. This approach has the advantage that it is much easier to pass the stack on to the following instruction than in the $\mathcal{E}$ scheme where we would need an attribute grammar to do so. Besides the

stack :: [value]
$\mathcal{A}$ :: [instr] → stack → [stack] → [instr]

$\mathcal{A}[\![\textbf{fun}\ name;\ \text{code}]\!]\ \text{S E}$	$= \textbf{label}\ name;\ \mathcal{A}[\![\text{code}]\!]\ \varepsilon\ (\text{S:E})$
$\mathcal{A}[\![\textbf{push}\ id;\ \text{code}]\!]\ \text{S E}$	$= \mathcal{A}[\![\text{code}]\!]\ (id\text{:S})\ \text{E}$
$\mathcal{A}[\![\textbf{pop}\ reg;\ \text{code}]\!]\ (v_1\text{:S})\ \text{E}$	$= \textbf{move}\ v_1\ reg;\ \mathcal{A}[\![\text{code}]\!]\ \text{S E}$
$\mathcal{A}[\![\textbf{pop}\ reg;\ \text{code}]\!]\ \varepsilon\ \text{E}$	$= \textbf{pop}\ reg;\ \mathcal{A}[\![\text{code}]\!]\ \varepsilon\ \text{E}$
$\mathcal{A}[\![\textbf{dup}\ n;\ \text{code}]\!]\ (v_1{:}\cdots{:}v_n\text{:S})\ \text{E}$	$= \textbf{move}\ v_n\ reg_{new};\ \mathcal{A}[\![\text{code}]\!]\ (reg_{new}{:}v_1{:}\cdots{:}v_n\text{:S})\ \text{E}$
$\mathcal{A}[\![\textbf{dup}\ n;\ \text{code}]\!]\ \text{S E}$	$= \textbf{fetch}\ (n-\#\text{S})\ reg_{new};\ \mathcal{A}[\![\text{code}]\!]\ (reg_{new}\text{:S})\ \text{E}$
$\mathcal{A}[\![\textbf{squeeze}\ n\ m;\ \text{code}]\!]\ (v_1{:}\cdots{:}v_{n+m}\text{:S})\ \text{E}$	$= \mathcal{A}[\![\text{code}]\!]\ (v_1{:}\cdots{:}v_n\text{:S})\ \text{E}$
$\mathcal{A}[\![\textbf{squeeze}\ n\ m;\ \text{code}]\!]\ (v_1{:}\cdots{:}v_n\text{:S})\ \text{E}$	$= \textbf{squeeze}\ 0\ (m-\#\text{S});\ \mathcal{A}[\![\text{code}]\!]\ (v_1{:}\cdots{:}v_n{:}\varepsilon)\ \text{E}$
$\mathcal{A}[\![\textbf{squeeze}\ n\ m;\ \text{code}]\!]\ \text{S E}$	$= \textbf{squeeze}\ (n-\#\text{S})\ m;\ \mathcal{A}[\![\text{code}]\!]\ \text{S E}$
$\mathcal{A}[\![\textbf{bfalse}\ lbl;\ \text{code}]\!]\ (v_1\text{:S})\ \text{E}$	$= \textbf{jfalse}\ v_1\ lbl;\ \mathcal{A}[\![\text{code}]\!]\ \text{S (S:E)}$
$\mathcal{A}[\![\textbf{bfalse}\ lbl;\ \text{code}]\!]\ \varepsilon\ \text{E}$	$= \textbf{bfalse}\ lbl;\ \mathcal{A}[\![\text{code}]\!]\ \varepsilon\ (\varepsilon\text{:E})$
$\mathcal{A}[\![\textbf{branch}\ fun;\ \text{code}]\!]\ (v_1{:}\cdots{:}v_s{:}\varepsilon)\ (\text{S}'\text{:E})$	$= \textbf{push}\ v_s;\ \cdots\ \textbf{push}\ v_1;\ \textbf{branch}\ fun;\ \mathcal{A}[\![\text{code}]\!]\ \text{S}'\ \text{E}$
$\mathcal{A}[\![\textbf{call}\ fun;\ \text{code}]\!]\ (v_1{:}\cdots{:}v_s{:}\varepsilon)\ \text{E}$	$= \textbf{push}\ v_s;\ \cdots\ \textbf{push}\ v_1;\ \textbf{branch}\ fun;\ \mathcal{A}[\![\text{code}]\!]\ \varepsilon\ \text{E}$
$\mathcal{A}[\![\textbf{return};\ \text{code}]\!]\ (v_1{:}v_2{:}\varepsilon)\ (\text{S}'\text{:E})$	$= \textbf{push}\ v_2;\ \textbf{branch}\ v_1;\ \mathcal{A}[\![\text{code}]\!]\ \text{S}'\ \text{E}$
$\mathcal{A}[\![\textbf{return};\ \text{code}]\!]\ (v_1{:}\varepsilon)\ (\text{S}'\text{:E})$	$= \textbf{branch}\ v_1;\ \mathcal{A}[\![\text{code}]\!]\ \text{S}'\ \text{E}$
$\mathcal{A}[\![\text{Instr};\ \text{code}]\!]\ \text{S E}$	$= \text{Instr};\ \mathcal{A}[\![\text{code}]\!]\ \text{S E}$
$\mathcal{A}[\![\varepsilon]\!]\ \text{S E}$	$= \varepsilon$

Figure 5.9: Compile-time stack optimisation.

instruction stream and the (simulated) stack the $\mathcal{A}$ scheme in Figure 5.9 takes an environment argument to record the lexical scope.

When translating the **dup** instruction the $\mathcal{A}$ scheme first checks whether the referenced stack item is present in the simulated stack or not. In the latter case a **fetch** instruction is issued to load the value from the physical stack into a fresh (virtual) KOALA register (reg_{new}). Otherwise the value is copied into a fresh register to avoid aliasing problems (see next section). In general the $\mathcal{A}$ scheme contains multiple rules for one KOALA instruction depending on whether the instructions arguments are present in the simulated stack or not.

An important invariant of the calling sequence in the $\mathcal{A}$ scheme is that parameters are passed on the physical KOALA stack. Therefore the $\mathcal{A}$ scheme flushes the simulated stack to the KOALA stack with a sequence of **push** instructions when calling a function (see the rules for **branch** and **call**). The same holds for **return**ing a result.

The Environment argument is used to handle the if-then-else construct of Functional C. Since the syntax of Functional C guarantees that each conditional branch terminates with a return statement, the lexical scope structure is a simple tree. The FCG compilation schemes traverse this tree in a fixed order (i.e. then before else), hence, we can record the lexical scopes with a stack. When the

$\mathcal{A}$ scheme enters the then branch, it stacks the current (simulated) stack for the else branch; the beginning of the then branch is marked by the **bfalse** instruction. When the $\mathcal{A}$ scheme enters the else branch it pops the saved stack from the environment argument; the end of the then part is marked by a **return** or **branch**.

The $\mathcal{A}$ scheme uses the $\mathcal{K}$ scheme from Figure 5.7 (augmented with Figure 5.8) as follows: $\mathcal{A}\,[\![\mathcal{K}\,[\![\text{prog}]\!]\,]\!]\;\varepsilon\;\varepsilon$. Compiling the *append* example results in registers being used, but the net effect is zero since the basic blocks do not contain any real work; just setting up stack frames to call functions does not benefit from register optimisations when parameters are passed on the stack. The new append code does not contain revision bars since it has changed too much:

```
label append;   fetch 2 R0;
                push R0;
                push L0;
                branch null;
label L0;       bfalse L1;
                fetch 3 R1;
                fetch 1 R2;
                squeeze 0 3;
                push R1;
                push R2;
                branch reduce;
label L1;       fetch 2 R3;
                push #0;
                push R3;
                push L3;
                branch field;
label L3;       fetch 3 R4;
                push #4;
                push R4;
                push L4;
                branch field;
label L4;       fetch 5 R5;
                fetch 1 R6;
                push R5;
                push R6;
                push prel_append;
                push L5;
                branch vap;
label L5;       fetch 3 R7;
                fetch 4 R8;
                squeeze 1 5;
                push R7;
                push R8;
                branch cons;
```

5.4.3 Inlining of primitive functions

A first solution to make the simulated stack of the $\mathcal{A}$ scheme more effective, is to enlarge the basic blocks by inlining some of the primitive functions. Many of these primitives map to a single **alu** instruction, if the operands are available in registers. The following example shows the typical coding style of such primitives for the null and field primitives:

fun null_2;	**fun** field_3;
pop reg_{ret};	**pop** reg_{ret};
pop reg_{list};	**pop** reg_{base};
alu eq reg_{list} NIL reg_{test};	**pop** reg_{off};
push reg_{test};	**alu** add reg_{base} reg_{off} reg_{addr};
branch reg_{ret};	**load** reg_{addr} reg_{val};
	push reg_{val};
	branch reg_{ret};

Figure 5.10: *Null* and *field* primitives in KOALA.

Inlining the primitive code in the KOALA instruction stream directly does not work for two reasons. Firstly, registers used in the primitives have to be renamed to avoid name clashes (α conversion). Secondly, the $\mathcal{A}$ scheme interprets the primitive's trailing **branch** instruction as a basic block marker, and flushes the simulated stack to memory, which reduces the benefits of the compile-time stack simulation. Therefore the additional inlining rules in Figure 5.11 use the α-converted-body() function that renames registers and strips the **fun** pseudo, the first **pop** and the last **branch** instruction of the primitive code. The second rule handles the tail call of a primitive function.

$\mathcal{A}[\![$**push** lbl; **call** $inline\_prim$; **label** lbl; code$]\!]$ S E
$\quad = \quad \mathcal{A}[\![\alpha$-converted-body($inline\_prim$); code$]\!]$ S E
$\mathcal{A}[\![$**dup** c; **squeeze** $n\ m$; **branch** $inline\_prim$; code$]\!]$ S E
$\quad = \quad \mathcal{A}[\![\alpha$-converted-body($inline\_prim$);
$\qquad$ **dup** $(c+1)$; **squeeze** 2 $(m+n-1)$; **return**; code$]\!]$ S E

Figure 5.11: Rules for inlining primitive functions, to be added to those in Figure 5.9.

The *append* function greatly benefits from inlining the *null* and *field* primitives. In reality all of the simple primitives are inlined, but this is not shown here for brevity.

```
label append;fetch 2 R0;
             move R0 R10;
             alu eq R10 NIL R11;
             jfalse R11 L1;
             fetch 3 R1;
             fetch 1 R2;
             squeeze 0 3;
             push R1;
             push R2;
             branch reduce;
label L1;    fetch 2 R3;
             move R3 R20;
             move #0 R21;
             alu add R20 R21 R22;
             load R22 R23;
             fetch 2 R4;
             move R4 R30;
             move #4 R31;
             alu add R30 R31 R32;
             load R32 R33;
             fetch 3 R5;
             move R33 R6;
             push R23;
             push R33;
             push R5;
             push R6;
             push prel_append;
             push L5;
             branch vap;
label L5;    fetch 3 R7;
             fetch 4 R8;
             squeeze 1 5;
             push R7;
             push R8;
             branch cons;
```

The apparent redundant data movement between registers will be handled by the KOALA assembler (i.e. the C compiler), and is of no concern for the $\mathcal{A}$ scheme. The basic blocks can be enlarged even further by inlining user functions. this could be done by the $\mathcal{A}$ scheme (two passes), but the FAST front end is already capable of inlining user functions.

5.4.4 Parameters passed in registers

Now that we have used the simulated $\mathcal{A}$ stack to optimise stack instructions inside basic blocks, we would like to extend the scheme to optimise parameter passing between functions as well. This is attractive since the callee can use its arguments directly from registers instead of loading them from the (physical) stack first. Such register parameters still have to be saved on the stack if the callee itself calls another function, except when it makes a tail call ($\approx 25\%$ of all calls). Quite often functions terminate by replying a value directly, in which case the parameters do not have to be saved at all. In general passing parameters in registers extends the basic blocks across function calls until the first sub function call and thereby provides more opportunity to optimise stack instructions.

The calling sequence will be changed as follows: parameters are passed in "global" registers, while the caller will save its internal state (arguments + locals) on the stack. When the caller resumes execution it will not restore the internal state in registers immediately, but rather fetch values from the stack on demand. This lazy scheme is advantageous after function calls if not all of

$\mathcal{A}[\![$**fun** $name_n$; code$]\!]$ S E	= **label** $name$;
	$\mathcal{A}[\![$code$]\!]$ (param$_1$:$\cdots$:param$_n$:ε) (S:E)
$\mathcal{A}[\![$**branch** *fun_n*; code$]\!]$ (v_1:$\cdots$:v_n:ε) (S$'$:E)	= **move** v_1 param$_1$; $\cdots$; **move** v_n param$_n$;
	branch *fun*;
	$\mathcal{A}[\![$code$]\!]$ S$'$ E
$\mathcal{A}[\![$**branch** *fun_n*; code$]\!]$ (v_1:$\cdots$:v_s:ε) (S$'$:E)	= **move** v_1 param$_1$; $\cdots$; **move** v_s param$_s$;
	pop param$_{s+1}$; $\cdots$; **pop** param$_n$;
	branch *fun*;
	$\mathcal{A}[\![$code$]\!]$ S$'$ E
$\mathcal{A}[\![$**call** *fun_n*; code$]\!]$ (v_1:$\cdots$:v_n:S) E	= **push** $v_{n+\#S}$; $\cdots$; **push** v_{n+1};
	move v_1 param$_1$; $\cdots$; **move** v_n param$_n$;
	branch *fun*;
	$\mathcal{A}[\![$code$]\!]$ ($reply$:ε) E
$\mathcal{A}[\![$**call** *fun_n*; code$]\!]$ (v_1:$\cdots$:v_s:ε) E	= **move** v_1 param$_1$; $\cdots$; **move** v_s param$_s$;
	pop param$_{s+1}$; $\cdots$; **pop** param$_n$;
	branch *fun*;
	$\mathcal{A}[\![$code$]\!]$ ($reply$:ε) E
$\mathcal{A}[\![$**return**; code$]\!]$ (v_1:v_2:ε) (S$'$:E)	= **move** v_2 $reply$; **branch** v_1; $\mathcal{A}[\![$code$]\!]$ S$'$ E
$\mathcal{A}[\![$**return**; code$]\!]$ (v_1:ε) (S$'$:E)	= **pop** $reply$; **branch** v_1; $\mathcal{A}[\![$code$]\!]$ S$'$ E

Figure 5.12: Calling sequence with parameter registers, replaces corresponding rules in Figure 5.9.

the internal state is used. A function result is also passed in a global register instead of on the stack.

Figure 5.12 implements the new calling sequence and replaces the **fun**, **branch**, **call**, and **return** rules in the previous $\mathcal{A}$ schemes. On function entry the simulated stack is no longer empty, but is loaded with the global parameters. On function exit, the result is moved to the *reply* register and control is passed back to the caller. A tail call (i.e. a **branch** instruction) is translated to a sequence of instructions that moves the call parameters (from the simulated stack) into the global *param*$_i$ registers; if not all parameters reside on the simulated stack then the remainder has to be fetched from the physical stack with **pop** instructions. Making a function call is slightly more complicated than the tail call case: if all parameters reside on the simulated stack then the additional stacked values (i.e. locals) have to be saved on the physical stack before transferring the parameters to their global registers, else the lacking parameters have to be fetched from the physical stack as with the tail call. The code after the call proceeds with a simulated stack that contains just the result value. If a reference is then made to the saved state, the $\mathcal{A}$ scheme will automatically fetch it from the physical stack. Now the code for *append* looks much better:

```
label append;move param2 R0;
             move R0 R10;
             alu eq R10 NIL R11;
             jfalse R11 L1;
             move param3 R1;
             move param1 R2;
             move R2 param1;
             move R1 param2;
             branch reduce;
label L1;    move param2 R3;
             move R3 R20;
             move #0 R21;
             alu add R20 R21 R22;
             load R22 R23;
             move param2 R4;
             move R4 R30;
             move #4 R31;
             alu add R30 R31 R32;
             load R32 R33;
             move param3 R5;
             move R33 R6;
             push param3;
             push param2;
             push param1;
             push R23;
             push R33;
             move L5 param1;
             move prel_apnd param2;
             move R6 param3;
             move R5 param4;
             branch vap;
label L5;    fetch 2 R7;
             fetch 3 R8;
             squeeze 0 5;
             move R8 param1;
             move R7 param2;
             move reply param3;
             branch cons;
```

5.4.5 Life-time analysis

The above calling sequence can be improved on two major points:

1. If after a function call the KOALA code makes multiple references to the same stack location, the $\mathcal{A}$ scheme will generate the same number of **fetch** instructions since the simulated stack is empty. One **fetch** instruction and some register moves provide the same functionality.

2. When calling a function, the $\mathcal{A}$ scheme blindly saves all local state on the physical stack, but often some stack locations will not be referenced in the remainder of the code. See for example the previous *append* code where after the *vap* call only two of the five saved values are being used.

Redundant loads from the stack can be avoided by adding another argument to the $\mathcal{A}$ scheme that records the status of the physical stack: if an item is fetched from the stack then its register is remembered.

The avoidance of saving dead variables is more difficult. Fortunately, the FAST front end has the ability to output pseudo function calls for a reference counting garbage collector, where they are used to increment or decrement the reference count of objects. The $\mathcal{A}$ scheme can take advantage of the increment/decrement pseudo functions by maintaining a life count with each item on the stack. When calling a function, only those stack items with a positive count have to be saved on the physical stack. A slight complication with this scheme is that the one-to-one correspondence between FCG's simulated stack locations and the actual physical location can no longer be maintained since we must not create holes in the stack, but in return we can reuse the stack locations of variables that were saved before and have died since the last function call.

We have constructed a new $\mathcal{O}$ptimal scheme that incorporates both improvements mentioned above. Since this $\mathcal{O}$ scheme is a straightforward extension of the previous $\mathcal{A}$ scheme, we have not provided a listing. The final *append* code no longer saves the unused locals when calling *vap*:

```
label append;move param2 R0;                load R32 R33;
             move R0 R10;                   move param3 R5;
             alu eq R10 NIL R11;            move R33 R6;
             jfalse R11 L1;                 push param1;
             move param3 R1;                push R23;
             move param1 R2;                move L5 param1;
             move R2 param1;                move prel_apnd param2;
             move R1 param2;                move R6 param3;
             branch reduce;                 move R5 param4;
label L1;    move param2 R3;                branch vap;
             move R3 R20;         label L5;fetch 1 R7;
             move #0 R21;                   fetch 2 R8;
             alu add R20 R21 R22;           squeeze 0 2;
             load R22 R23;                  move R8 param1;
             move param2 R4;                move R7 param2;
             move R4 R30;                   move reply param3;
             move #4 R31;                   branch cons;
             alu add R30 R31 R32;
```

If the above KOALA code is translated into C and fed to the GNU gcc compiler, all redundant register moves are eliminated from the append code. The generated assembly code for a SPARC processor is given below:

```
_append: cmp %l2,10                    !param2 == NIL ?
         bne L1                        !
         nop                           !branch delay slot
         b _reduce                     !call _reduce
         mov %l0,%l2                   !move param3 param2
l1:      ld [%l2],%l4                  !load head-field of param2
         st %l3,[%l1]                  !push param1
         st %l4,[%l1-4]                !push R23
         add %l1,-8,%l1                !adjust stack pointer
         mov %l0,%l4                   !move param3 param4
         ld [%l2+4],%l0                !load tail-field of param2
         sethi %hi(_prel_apnd),%l2     !load address of
         or %lo(_prel_apnd),%l2,%l2    !function _prel_apnd
         b _vap                        !call _vap, use delay slot
         mov 20,%l3                    !put return addr in param1
L5:      ld [%l1+8],%l2                !fetch 2 param1
         ld [%l1+4],%l3                !fetch 1 param2
         add %l1,8,%l1                 !squeeze 0 2
         b _cons                       !call _cons, use delay slot
         mov %l4,%l0                   !move reply param3
```

5.5 Performance

To assess the runtime performance effects of the optimisations described in the previous section, we have run a set of benchmark programs several times on a SUN 4/690. The majority of programs are sequential versions of the benchmark programs of Chapter 4.5: FFT, WANG, 15-PUZZLE, SCHED, COMP-LAB, and WAVE (see Table 4.1 on page 85). In addition the following three other serious applications have been used:

program	#lines	description
SOLID	605	Point membership classification algorithm of solid modeling library for computational geometry [Davy92].
TYPECHECK	360	Polymorphic type checking of a set of function definitions and printing of the type signatures [Peyton Jones87b, Chapter 9].
TRANSFORM	834	Transformation of 9 programs represented as synchronous process networks into master/slave style parallel programs [Vree92].

	naive	*+inline*	*+stack*	*+tailc*	*+params*	*+life/death*
FFT	2.3	1.7	1.2	1.1	1.0	1.0
WANG	8.8	6.3	4.9	4.6	4.1	4.0
15-PUZZLE	36.9	21.9	15.6	14.5	12.1	11.9
SCHED	34.9	23.8	16.0	13.5	12.0	11.4
COMP-LAB	6.9	4.6	3.2	3.0	2.7	2.5
WAVE	3.9	2.4	1.7	1.6	1.4	1.3
SOLID	38.5	24.7	16.5	15.7	14.7	14.0
TYPECHECK	45.8	28.4	19.0	16.4	14.0	13.2
TRANSFORM	–1–	–1–	3.9	3.5	3.2	3.2

–1– GNU compiler runs out of memory.

Table 5.1: Execution time [sec] under various compiler optimisations.

The programs have been timed on a UNIX system using `/bin/time`, taking the sum of user and system time as the total execution time. Each program has been run 10 times in a row, on a quiet system, taking the best execution time as shown in Table 5.1.

The column marked naive contains the results for code that was produced by FCG with the straightforward $\mathcal{K}$, $\mathcal{R}$, and $\mathcal{E}$ schemes from Figure 5.7. The following columns list the results for adding the optimisations of the previous section one by one to the naive version: *i*nlining of primitives, *s*tack simulation, *t*ail call optimisation, *p*arameter registers, and *l*ife-time analysis. For example, the +*t* column presents the results for FCG with the basic schemes, the rule to inline primitives (Figure 5.11), the stack simulation of the $\mathcal{A}$ scheme (Figure 5.9), and the additional rule for tail calls (Figure 5.8).

As can be seen from the results, the optimisations improve the performance of the compiled code considerably. The largest difference is reached for the TYPECHECK program: the optimal (*+life/death*) version runs 3.5 times as fast as the naive version. The WANG program shows the smallest improvement under the various optimisations: only a factor 2.2. As is apparent from the results, the inlining of the primitive functions and stack optimisation are of vital importance for generating quality code, while the additional exploitation of registers has a surprisingly low effect of less than 25% performance increase. Note, however, that it is rather difficult to assess the effects of individual optimisations by comparing two columns in Table 5.1; the various optimisations effect each other, for example, the results of simulating the argument stack at compile time heavily depend on the inlining of primitive operations, otherwise all arguments have to be passed via the stack and nothing is gained at all.

To judge the absolute performance of the code generated by FCG, we have made a comparison with several other state-of-the-art lazy functional language compilers: the Concurrent Clean compiler from Nijmegen University [Nöcker91a, Smetsers91], the LML compiler developed at Chalmers University [Augustsson89a, Augustsson90], the Haskell compilers of Chalmers University and Glasgow University, and the original code generator for the FAST compiler [Hartel91b]. To avoid the tedious and error prone work of converting the Miranda programs into the other languages the FAST front end has been adapted to either produce an executable or, depending on a compiler switch, a Haskell, LML, or Clean source program. Considerable effort has been put in generating "equivalent" programs for those languages, see [Hartel93].

Table 5.2 lists the execution times of the benchmark programs compiled by each of the six compilers. All figures are the minimum user+system time of 10 runs of the executable with 16 Mbyte heap space on a quiet UNIX system.

language	Clean	FAST	FCG	LML	Haskell	
compiler					Chalmers	Glasgow
version	0.8.1	29	3	0.998	0.998	0.10
Compilation speed in lines per minute real time						
minimum	* 354	74	13	126	69	29
maximum	* 1113	191	173	291	216	99
Execution time in seconds						
FFT	10.8	2.0	* 1.0	2.2	5.1	4.1
WANG	* 2.9	9.9	4.1	4.5	4.6	3.3
15-PUZZLE	11.3	40.5	11.8	16.2	13.3	* 9.5
SCHED	18.8	–2–	*11.6	17.2	18.2	11.4
COMP-LAB	* 2.4	6.3	2.6	2.9	3.8	3.2
WAVE	9.4	3.7	* 1.3	7.8	17.8	12.9
SOLID	17.1	28.8	*14.3	26.7	21.3	–1–
TYPECHECK	*12.5	33.9	13.6	15.4	16.2	13.1
TRANSFORM	4.0	6.9	* 3.1	3.3	3.4	3.7

* Best execution time.
–1– segmentation fault in the compiler.
–2– runs out of heap space.

Table 5.2: Benchmark results showing execution times in seconds for runs with 16Mb of heap space on a SUN 4/690 with 64Mb of real memory and 64Kb cache.

In addition the compilation speed is reported in lines per minute real time; for each compiler the minimum and maximum speed is reported, as found over the whole range of benchmark programs. Each row bears one asterisk, which marks the best result for that particular row. This shows that it depends to some extent on the application which compiler generates the fastest code.

The comparison between FCG and FAST is especially interesting since both compilers use the same front end that generates Functional C. In contrast to FCG, the FAST compiler directly feeds Functional C to the GNU C-compiler; to achieve acceptable performance, a header file is included that contains macro definitions for simple primitives like integer addition, null test, etc. The benchmark results show that the unoptimised FCG version (column naive in Table 5.1) generates code that runs at similar speed as programs compiled by FAST, while the optimised FCG compiler (Table 5.2) generates code that runs two to three times as fast as FAST. This is a rather surprising result since FAST does not reclaim garbage and FCG generates extra code to manipulate tag bits that are present in each data value to support garbage collection. Apparently this overhead is of no great importance since FCG also outperforms the other state-of-the-art compilers on most programs.

The exceptional performance of FCG (and FAST) on the FFT and WAVE programs is caused by the efficient array support through built-in primitives. Although both applications use the Haskell style arrays [Hudak92], the LML and Haskell(!) implementations just outperform the Clean programs that use lazy lists, but do not match performance of the FCG array primitives. For the other benchmark applications, which do not use arrays, the difference in code quality generated by various compilers is much smaller.

The FAST, FCG and Glasgow Haskell compilers generate C programs, while the remaining compilers generate assembly directly. It is interesting to note that using C as a portable high-level assembler does not mean generating bad code. Considerable optimisation and tuning, however, is required to produce C programs that the C compiler properly understands. Apparently the Glasgow Haskell compiler has not reached the level of sophisticated optimisations presented in Section 5.4 because of the complexity of the full Haskell language. Unfortunately, using a C compiler instead of an assembler to produce object code increases compilation time significantly. This is especially true for the FCG compiler that generates one C function containing all "assembly" code. Note that this approach also rules out separate compilation. The Clean compiler generating native assembly is much faster than all other compilers.

5.6 Conclusions

We have built an efficient, portable, functional language compiler that supports moving garbage collectors with minimal effort. This has been accomplished by reusing large parts of existing compiler technology: the FAST front end for making lazy evaluation explicit, and the C compiler for generating optimised code.

The FCG code generator consists of two passes that are described with simple transformation schemes. The first basic scheme describes a recursive descent parser that generates code for a pure stack machine. This code is optimised by the second scheme that makes a linear scan over the code to combine matching stack instructions into register based equivalents. Both schemes can be combined into one attribute grammar, but that would make the optimisation scheme far more difficult to understand since the inherently sequential state information flow has to be propagated indirectly through the parse tree.

The performance results of a benchmark of functional programs, show that the optimisations have a large effect; the difference between naive code and the optimised version ranges between a factor 2.2 and 3.4. The comparison between the FAST compiler, which does not perform garbage collection, and FCG, which includes a copying collector, shows that FCG outperforms FAST on all benchmark programs. The benchmark results of the Clean, LML, and Haskell state-of-the-art compilers show that FCG generates quality code that often performs best, especially if arrays are being used, but at the price of low compilation speed.

Because of the clear separation between graph reduction and parallelism in the WYBERT design and the tagged data representation of graph nodes, the FCG compiler can be used without change to generate code for execution on a parallel machine.

Chapter 6

Experimental results†

This chapter provides measurements of the performance of the integrated WYBERT system on real hardware. The performance results of a set of benchmark programs is used to asses the significance of the advantages of the WYBERT approach. For example a comparison with a standard parallel implementation technique shows the performance gain obtained by not locking nodes in shared memory during ordinary graph reduction. The effects of the special resource management policies of WYBERT are analysed by a performance monitoring tool that provides a detailed cost break down of the execution time of an application.

Benchmark programs written in Miranda and annotated with the sandwich construct are compiled with the FAST/FCG compiler (Chapter 5) and linked with a straightforward implementation of the runtime support system as discussed in Chapter 4. Then the object code is down-loaded for execution onto a Motorola *HYPERmodule* that consists of four MC88100 RISC processors, equipped with 32Kbyte caches each, connected to 64 Mbytes of shared memory (see Figure 6.1). The combination of a state-of-the-art compiler with advanced hardware gives us today's fastest parallel implementation of a lazy functional language as already stated in the survey of Chapter 3.

A number of experiments have been conducted to study the individual impact in the WYBERT system of the FRATS reduction strategy, LIFO/FIFO scheduling, storage management (BAS vs. VAS), and garbage collection. To easily compare the various design alternatives, these experiments work with small input sets for the following seven benchmark programs: NFIB, QUEENS, DET, WANG, 15-PUZZLE, COMP-LAB, and WAVE; for a short description see Table 4.1

†This chapter represents joint work with Henk Muller.

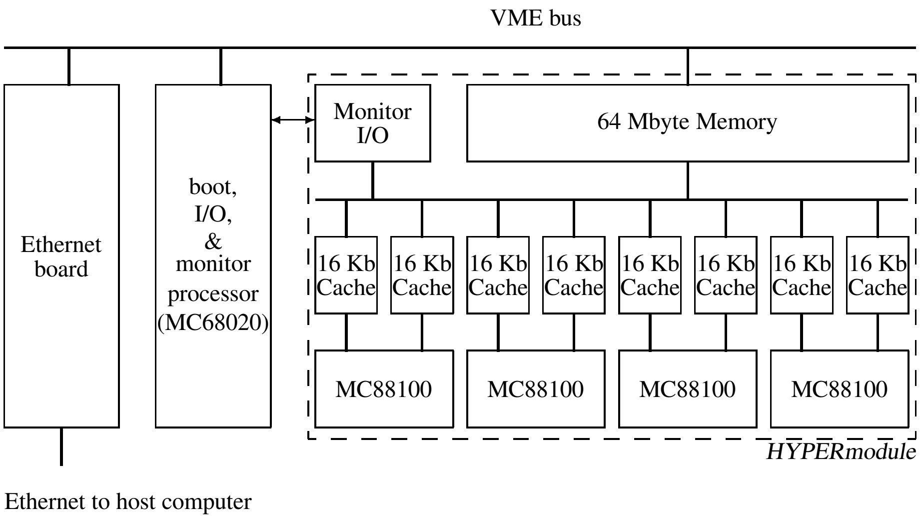


Figure 6.1: Hardware configuration: VME crate with ethernet board, I/O controller, and *HYPERmodule*.

on page 85. At the end of this chapter, a set of large input parameters is used to show the behaviour of these benchmark programs in a realistic setting by giving a detailed break down of the execution costs. Table 6.1 lists the key properties of the benchmark programs for both parameter sets.

Simple divide-and-conquer applications like NFIB and QUEENS unfold into a plain tree shaped task structure. Multi pass applications generate a chain of

		small		large	
	task structure	av par	node claim	av par	node claim
NFIB	tree	75.9	0.011 Mb	198.9	0.03 Mb
QUEENS	tree	69.2	1.3 Mb	97.7	37 Mb
DET	2-d spine	51.5	8.0 Mb	89.8	750 Mb
WANG	chain, length 2	11.6	11.7 Mb	11.9	67 Mb
15-PUZZLE	chain, length 3	29.7	18.3 Mb	49.7	318 Mb
COMP-LAB	chain, length 2	9.4	5.6 Mb	14.5	72 Mb
WAVE	chain, length n	5.6	23.3 Mb	5.0	211 Mb

Table 6.1: Benchmark properties for small and large input sets.

successive unfolding/folding task trees; WANG, 15-PUZZLE, and COMP-LAB generate a chain of fixed length, while WAVE generates a chain of length "number-of-iterations". The *average parallelism* in Table 6.1 indicates the useful parallelism of an application; it is the maximum speed-up that can be achieved with an unlimited number of processors [Eager89]. The amount of space allocated for graph nodes in the heap is given in the columns marked "node claim".

6.1 FRATS reduction strategy

The FRATS reduction strategy for the sandwich annotation squeezes shared redexes out before sparking tasks for parallel execution. As a consequence tasks do not share redexes in the heap at runtime, so locking of graph nodes is unnecessary. Garbage can be collected per task and reduction stacks can be merged efficiently into one stack per processor (see Chapter 4). A disadvantage of FRATS is that the eager evaluation of potentially shared expressions might result in superfluous, or even non terminating, computations. This problem is solved by applying program transformations as shown in Section 4.2. The current set of benchmark programs has been fine-tuned* and therefore incur neglectable squeeze overhead as will be shown in Section 6.5.4.

The SIS simulator used in Section 4.2 does not take low-level details into account, and thus cannot determine WYBERT's benefit of not locking each and every application node during graph reduction. To measure the costs of locking, we have constructed another parallel implementation of the sandwich annotation based on the common spark-and-wait model. This implementation will be described in the next section, and is used in Section 6.1.2 to quantify the advantage of FRATS over parallel implementation methods of lazy functional languages that use locking.

6.1.1 Spark-and-wait implementation

The general spark-and-wait model for generating and managing parallel tasks has been described in Section 3.1.1. Instead of rewriting the complete set of benchmarks to replace the sandwich annotation by spark-constructs, the spark-and-wait model is implemented as another library of runtime support functions. The spark-and-wait implementation of the sandwich construct does not squeeze task arguments, but protects application nodes of concurrent access through the *xmem*-instruction of the MC88100 processor; whenever the graph reducer

*Thanks to Rutger Hofman.

needs to evaluate a delayed computation it atomically exchanges the function pointer in the suspension node with a pointer to the wait function. If another reducer tries to evaluate the same suspension node, it automatically invokes this wait function, which suspends the current task and returns control to the scheduler. Instead of placing the suspended task on a waiting list associated with the suspension node, the spark-and-wait scheduler employs a polling mechanism: whenever the scheduler looks for work it first checks whether or not the top-of-stack task has become unblocked because the suspension node has been overwritten by an indirection node holding the requested value.

Since the task expressions are not specially marked, the graph reducer automatically invokes the execution of a not yet evaluated task whenever it needs that task's value, i.e. the evaluate-and-die model is being used (see Section 3.1.5). When handing out a task for execution, the spark-and-wait scheduler first removes tasks from the ready pool whose corresponding graph node has already been overwritten to avoid initialisation overhead. Often the task value has already been computed (by the parent), or some reducer has already started the evaluation; in the first case the graph node is overwritten by an indirection node, while in the latter case the node is overwritten by the wait-function description. Because of the difference in task synchronisation and the presence of shared redexes, spark-and-wait programs running under control of the standard ToS scheduler can deadlock; the task dependency graph is not restricted to a tree as in case of WYBERT, but has an arbitrary acyclic shape. Therefore the ToS scheduler of WYBERT has been adapted to select a ready task somewhere in the stack, and to copy the context on top of the stack before resuming execution. The copy costs are small since measurements have shown that the context of a task is between 8 and 163 words for the benchmark programs.

Time has prohibited to implement a global garbage collector for the spark-and-wait implementation, so all experiments have been arranged to run without a single collect for both the sandwich and spark-and-wait implementation.

6.1.2 Sandwich versus spark-and-wait

The benchmark programs with small input sets have been used to compare WYBERT, which squeezes shared redexes in advance, with the spark-and-wait implementation that locks shared redexes as part of the graph reduction process. The measurements reported in Table 6.2 give the ratio of execution times of the WYBERT implementation over the spark-and-wait implementation. It shows

	#processors			
	1	2	3	4
NFIB	1.00	0.99	1.01	0.99
QUEENS	1.00	1.00	0.99	1.00
DET	0.87	0.87	0.79	0.74
WANG	0.86	0.75	0.68	0.58
15-PUZZLE	0.93	0.89	0.84	0.78
COMP-LAB	0.96	0.82	0.76	0.75
WAVE	0.93	0.64	0.57	0.49

Table 6.2: The ratio of WYBERT versus spark-and-wait execution times.

that for programs that rarely use lazy evaluation, like NFIB and QUEENS, the small number of *xmem* instructions has only a small impact on the difference in execution time of both implementations (i.e. a ratio of 1.0). However, for large applications like WANG and 15-PUZZLE, which often invoke suspended computations stored in the heap, the avoidance of locking is substantial. For example, the WAVE program under WYBERT takes half the execution time of its spark-and-wait counterpart on the full *HYPERmodule*.

When increasing the number of processors, the gain in performance of WYBERT by eliminating locking overhead increases in comparison to the spark-and-wait implementation. The locking of graph nodes in case of spark-and-wait on a multiple processor system does not only affect the performance of the local processor as they execute additional instructions, but influences the others as well. Each *xmem* instruction writes the new value through the cache resulting in additional contention on the shared bus. Furthermore all caches have to "snoop" this write to maintain cache coherency and keep their cached data consistent. Whenever a cache snoops a memory transaction it stalls the processor read or write since it cannot look up the status of two addresses in parallel, hence, a single *xmem* instruction effectively slows down each processor in the shared-memory machine. The slow down is not a simple function of the number of executed *xmem* instructions, since the effect of contention on the memory bus and the impact of snooping depend on the exact state of the complete system when an *xmem* is executed. Therefore we have not listed the number of *xmem* instructions per processor, but only the overall effect on execution time. The spark-and-wait implementation does suffer from the slow down caused by *xmem* instructions as can be seen in the measurements as given in Table 6.2. For example, WYBERT performs 22% better than spark-

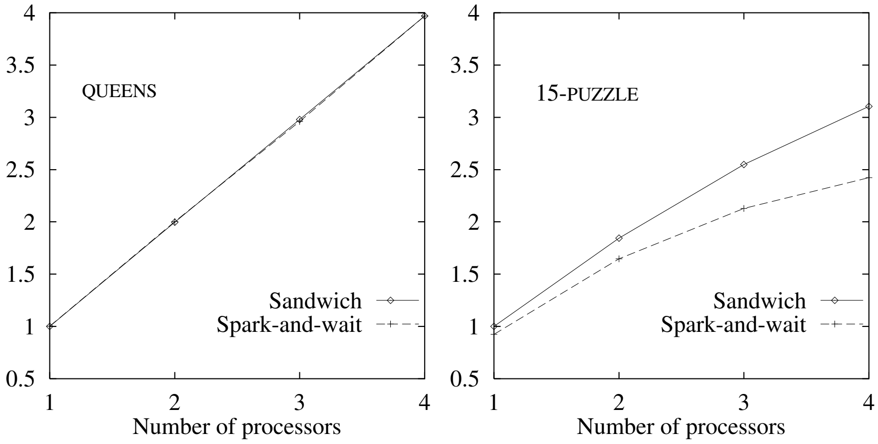


Figure 6.2: Speed-up curves for WYBERT and spark-and-wait.

and-wait for the 15-PUZZLE on a four processor system, while it only performs 7% better on a single node configuration.

WYBERT itself suffers also from external cache requests since each miss of a local processor results in a memory transaction that has to be snooped by all other caches in the system for consistency. This effect can be seen in Figure 6.2 where the speed-up curves have been drawn for two typical applications. The speed-up is computed with respect to the single processor execution time including task management overhead. All measurements exclude the time to down-load an executable and initialise the input data set. The QUEENS program uses few heap nodes and therefore issues a modest number of *xmem* instructions. As a consequence the QUEENS application shows the same perfect linear speed-up for both WYBERT and spark-and-wait. The 15-PUZZLE that uses a large number of heap cells shows just a speed-up of 3.2 under WYBERT on four processors. This is caused by cache conflicts between processor and memory accesses, and by sequential execution inherent to the functional program itself. Section 6.5.4 provides measurements of the percentage of idle time for several applications.

6.2 Scheduling

The ToS scheduler of WYBERT allows for efficient stack allocation: all tasks assigned to one processor share a single processor stack, which is used as a stack of stacks. As only the topmost task on the stack can execute, the scheduler is limited in selecting a task for execution. In Section 4.3 it was shown by simulation that the Top-of-Stack (ToS) constraint has little impact on performance in comparison to general list scheduling policies. To verify these positive findings, the ToS scheduler of WYBERT has been modified to account for what fraction of execution time is spent running idle while some task ready for execution is blocked because of the ToS constraint. Measurements showed that less than 0.5% of the execution time is lost on the four processor *HYPERmodule* implementation, hence, we have not bothered to build a parallel implementation with a stack per task model.

	#processors			
	1	2	3	4
NFIB	1.00	1.01	1.00	1.01
QUEENS	1.00	1.01	1.00	1.01
DET	1.00	1.01	1.02	1.00
WANG	1.00	1.00	1.01	1.00
15-PUZZLE	1.00	1.00	1.00	1.01
COMP-LAB	1.00	0.91	0.99	0.97
WAVE	1.00	0.98	0.98	1.02

Table 6.3: The execution-time ratio of ToS with local versus global task pools.

The simulation studies of ToS behaviour in Section 4.3 show that a scheduler based on a local task pool at each processor has advantages over a scheduler using a single global task pool shared by all processor. The default setting of the WYBERT scheduler is to use local task pools, but it can be changed to using a single global task pool. The hypothesis that a local task pool strategy outperforms a single global pool is not supported by the experimental results as listed in Table 6.3. The ratio of execution times is always close to 1.0, which indicates that both scheduling policies have similar performance. Sometimes the local pools give the best results; sometimes a global pool performs best. This can also be seen in the two example speed-up curves in Figure 6.3. Apparently the potential improvement in task scheduling does not outweigh the costs of maintaining multiple task lists, at least for small multiprocessors.

A difference between scheduling with local and global task pools, which does not show in the average performance results, is the variation in execution times measured over a number of runs. In case of the local task pool strategy execution times measured on the prototype are equal for different runs of the

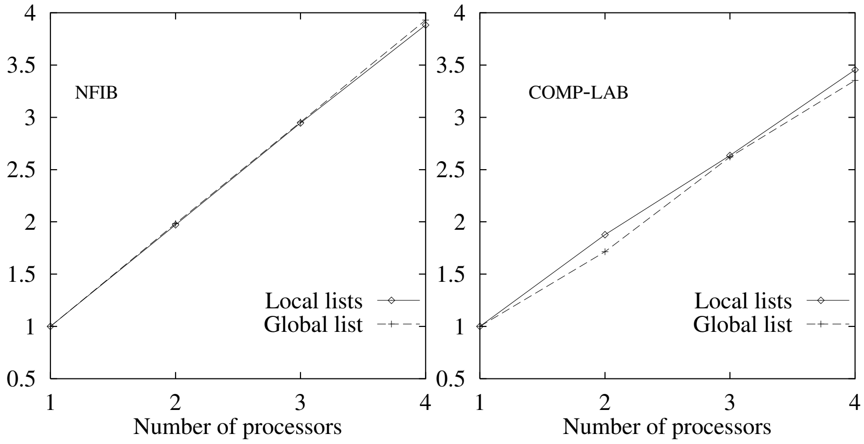


Figure 6.3: Speed-up curves for ToS with local and global task pools.

same application, but for the global task pool different execution times are measured for identical application runs. This shows that the behaviour of the global scheduling policy is rather dependent on which processor acquires access to the global pool first in case of conflicts on the single semaphore. This results in a highly unpredictable assignment of tasks to processors, while the local task pools give rise to rather deterministic schedules. This "predictable" behaviour of scheduling with local pools is an advantage for debugging and performance modelling.

The experiments do not show a performance advantage of local tasks pools over a single global pool, but it can be expected that the shared pool becomes a bottleneck in large shared-memory multiprocessors. The advantage of local task pools has been observed in the SIS simulation studies of Section 4.3, but it requires a larger machine than the four node *HYPERmodule* to confirm these findings in practice.

6.3 Storage management

To support efficient local garbage collection, the WYBERT storage manager allocates memory blocks such that the heap blocks of a task never interleaves with those of its ancestors (see Section 4.4). Therefore garbage of a task can be reclaimed with an ordinary two-space compacting garbage collector that operates independently of other tasks and processors. In Section 4.4

three memory management policies have been presented that enforce the strict separation of heaps: BAS, VAS, and CAS. The Basic Allocation Scheme simply allocates a new block above the last allocated block in the virtual addresses space without reusing any of the blocks freed after a garbage collect. To make better use of the of virtual address space, the Virtual Allocation Scheme reuses virtual addresses space on the fly by allocating a new block in the first free address range above the heap of the task's parent. The Circular Allocation Scheme operates analogous to VAS but implements virtual address space entirely in software by explicitly controlling the most significant address bits.

	#processors			
	1	2	3	4
NFIB	1.00	1.00	1.00	1.00
QUEENS	1.00	1.00	1.00	1.00
DET	1.04	1.06	1.08	1.07
WANG	1.02	1.03	1.03	1.04
15-PUZZLE	1.03	1.05	1.06	1.08
COMP-LAB	1.02	1.03	1.01	1.02
WAVE	1.02	1.03	1.02	1.02

Table 6.4: Performance ratio of VAS versus BAS memory allocation.

The simple BAS strategy rapidly consumes virtual address space and cannot handle large applications like the experiments that will be described in Section 6.5.4. The problem can be solved by implementing a complex virtual address space compactor, but this is an expensive operation. The MiG simulations have shown that VAS does not need such compactions if the virtual address space (2 Gbyte) is at least three times the physical amount of memory (64 Mbyte). CAS always needs compactions to recover from fragmentation. For these reasons the VAS strategy has been chosen as the memory management policy for WYBERT.

To support the choice of VAS over BAS, we have implemented both strategies, without virtual address space compaction, using the MC88200 combined MMU and cache chip. Again the benchmark programs with small input sets have been used to compare both memory management policies, since BAS can not handle applications that claim more than 2 Gbyte of virtual address space. According to the MiG simulation studies, the VAS strategy is somewhat more expensive than BAS as can be seen in Table 6.4: the additional bookkeeping

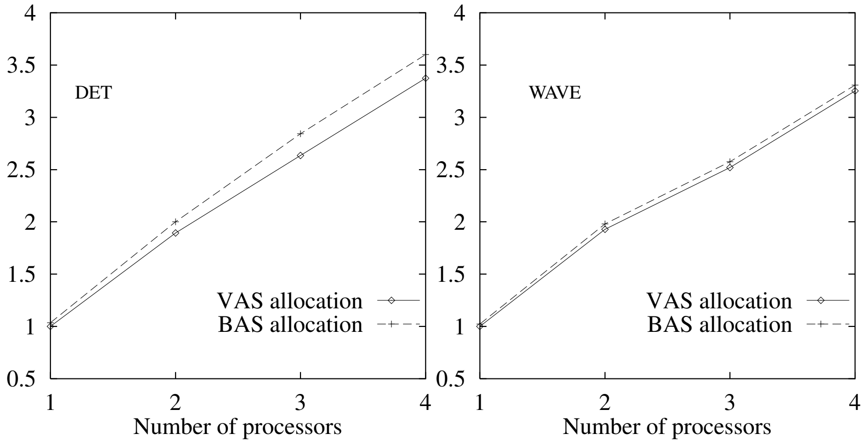


Figure 6.4: Speed-up curves for VAS and BAS policies.

results in up to 8% difference in execution time (15-PUZZLE). The comparison, however, holds only for these small applications since BAS would incur significant overhead to compact its address space in case of large programs. A compaction of the virtual address space requires the relocation of all pointers, hence, the total amount of physical space in use (max. 64 Mbytes) has to be read and written. This will take approximately 10 seconds for each 50 seconds of computation time.

Two example speed-up curves are shown in Figure 6.4. The speed-up curves for DET show that the VAS overhead increases for larger number of processors. This indicates that the central bookkeeping (e.g., page table maintenance) is a bottleneck in the current (unoptimised) implementation.

To minimise memory fragmentation, VAS has been adapted to allocate memory in blocks whose size is smaller than the hardware dictated 4Kbyte page size. The additional book keeping, however, is not for free as can bee seen in Table 6.5 where the execution times of the benchmark for varying block sizes are compared to the standard case of 4Kbyte that equals the hardware supported page size. None of the applications benefit from small block sizes. Apparently, the advantage of less memory fragmentation does not outweigh the additional overhead in bookkeeping.

For "real" divide-and-conquer applications with coarse grain tasks (see Section 6.5.4), the number of (waiting) tasks is small. Hence, the amount of space wasted due to fragmentation is small, so VAS can be used with a large block size (i.e. 4Kbyte) to limit the allocation overhead.

	block size [bytes]				
	256	512	1K	2K	4K
NFIB	1.00	1.00	1.00	1.00	1.00
QUEENS	1.01	1.00	1.00	1.00	1.00
DET	2.25	1.56	1.23	1.05	1.00
WANG	1.58	1.22	1.07	1.02	1.00
15-PUZZLE	–1–	1.48	1.24	1.10	1.00
COMP-LAB	1.21	1.09	1.04	1.00	1.00
WAVE	–1–	1.09	1.04	1.02	1.00

–1– shortage of heap memory because of excessive bookkeeping.

Table 6.5: Performance ratio of VAS for various block sizes vs. standard 4Kbyte page size.

6.4 Garbage collection

Garbage collection is an important aspect of practical functional language implementations, be it sequential or parallel, since graph reduction allocates new heap cells at a high speed. Even the 64 Mbyte of shared memory in the prototype machine is consumed in less than a minute real time by an average benchmark application.

The VAS storage management strategy of WYBERT allocates heaps such that leaf tasks can collect their garbage independently of other tasks and processors. This avoids the need for a system-wide synchronisation of all processors to join in a global garbage collect where individual nodes have to be locked to maintain sharing and enforce consistency. Another benefit of WYBERT is that tasks can time share a common to-space, so not half the available memory has to be reserved for the to-space buffer, but only one fifth. Hence more memory can be used for graph reduction: 48 Mbyte instead of only 30 Mbyte. To minimise the performance degradation when several processors attempt to garbage collect at the same time, the busy-wait loop on the semaphore that guards the single to-space has been explicitly coded to test the semaphore before trying to grab it with an *xmem* instruction. This results in an instruction loop that repeatedly reads the semaphore value without modifying it, hence, the value will be cached locally at each busy-waiting processor without causing traffic on the memory bus, so active processors can proceed at full speed.

To measure the garbage collection overhead in WYBERT, we compared a version of the runtime support system with and a version without garbage collection. In case of garbage collection the available heap space has been set to a value 20% above the absolute minimum needed to run to completion. This assures that a “reasonable” number of garbage collects is performed. In Section 6.5.4 the garbage collection overhead will be given for realistic applications that have the full 64 Mbyte of the *HYPERmodule* at their disposal.

	#processors			
	1	2	3	4
NFIB	1.00	1.00	1.00	1.00
QUEENS	1.00	1.00	1.00	1.00
DET	1.02	1.01	1.00	0.98
WANG	1.08	1.09	1.11	1.11
15-PUZZLE	1.03	1.02	1.00	0.98
COMP-LAB	1.07	1.09	1.00	1.03
WAVE	1.25	1.25	1.32	1.23

Table 6.6: Ratio of execution times of applications with and without (local) garbage collection.

Both the numbers in Table 6.6 and speed-up curves in Figure 6.5 show that garbage collection overhead is small for all programs except WAVE. In some cases the version with garbage collection even outperforms the implementation without. This is most likely caused by caching effects; the garbage collector compacts live data into a contiguous block of heap space, hence, improves spatial locality.

The WAVE program shows exceptional behaviour in comparison to the other applications: 25% - 32% increase in execution time when garbage collection is performed. This is caused by the shared to-space in combination with the regular task structure of the WAVE program itself. The program unfolds into sub problems of equal length and behaviour. All four processors receive a task, start reducing, and at roughly the same moment decide that it is necessary to perform a local garbage collection. The single to-space, however, serialises the garbage collections, which could be performed in parallel with graph reduction otherwise, and results in processors running idle. Once the processors are running out of phase the remaining garbage collects do not clash anymore; modifying the program to spark fewer tasks would increase the execution time since fewer garbage collects would occur in parallel with graph reduction.

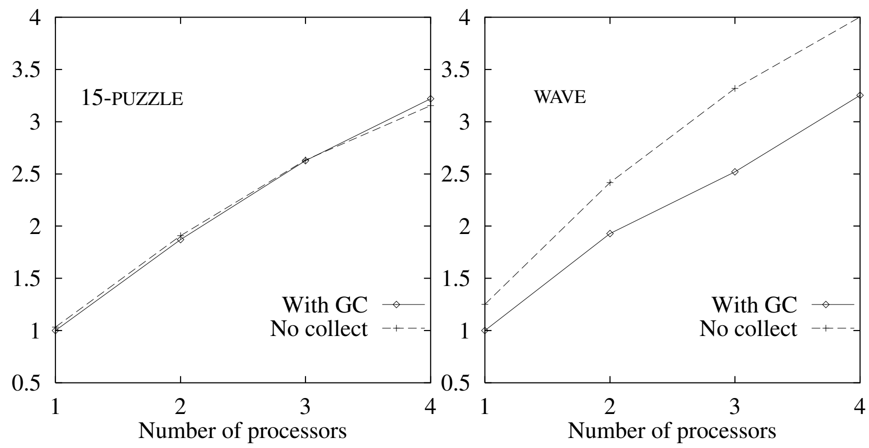


Figure 6.5: Speed-up curves for applications with and without garbage collection.

The number of to-spaces can be adjusted in the runtime support system, trading memory space for (potential) blocking on a to-space. Table 6.7 shows that the WAVE program performs even worse when the single to-space is replaced by four to-spaces to remove the garbage collection bottleneck. This decrease in performance is caused by the increased number of collects needed because of less available heap space. Only in case of two to-spaces, the WAVE program benefits from parallel garbage collections.

	1	2	3	4
WAVE	1.00	0.97	1.29	1.38

Table 6.7: Relative performance of WAVE with multiple to-spaces.

6.5 Execution profiling

In previous sections various design aspects of WYBERT have been studied individually by comparing execution times of alternative solutions. To deepen our understanding of individual application behaviour, a profile tool has been developed for the WYBERT system that measures where an application spends its execution time. During execution each processor maintains a global status field describing the current activity, which is sampled each millisecond by the boot-I/O processor via the common VME bus (see Figure 6.1). When the program has completed execution, the sampled status data, which has been buffered in 2 Mbyte internal memory on the boot-I/O card, is transmitted via ethernet to the host for analysis. Currently, the following activities are monitored:

user The processor is executing user code, i.e. it is performing graph reduction.

rts The processor is executing WYBERT's runtime support code for scheduling (ToS) and memory allocation (VAS).

gc The processor is running the two-space copying garbage collector.

idle The processor is busy waiting for either a new task to arrive, or the termination of a child task on another processor.

The potential perturbation of the system behaviour caused by the profiler is very small: one memory write per system call to record the status change and one memory read per millisecond on the VME bus to sample the current status. Therefore the effect of this low resolution profiling on the system can safely be neglected. In the sequel a number of example execution profiles will be presented that plot activity versus time of some typical programs. The aggregate cost break-down for the complete benchmark with realistic input sets will be provided in Section 6.5.4.

6.5.1 Queens

The first example profile plot is of the QUEENS program, see Figure 6.6. The QUEENS program is a representative of the class of easy parallelisable applications, which includes the NFIB and DET programs also. The program unfolds through some levels of recursion into a task tree with coarse grain leaf tasks. These coarse grain tasks take up the major part of execution time and can be scheduled for execution on one of the four processors without any constraint. Hence, the QUEENS program achieves a speed-up of 3.94 on the four processor *Hypermodule*.

An execution profile consists of two parts: the processor activity graphs and the system activity graphs. The processor activity graphs (the top four plots in Figure 6.6) show the status of each processor during the execution of the application: **user**, **rts**, **gc**, or **idle**. All processors start off **idle**, then grab a task and start reducing (state = **user**) invoking system calls to allocate more memory or to get the next task (state = **rts**). The system calls appear in the processor activity graphs as spikes between the **user** and **rts** activity lines. Near the end processors 1 and 3 run idle because no more new tasks are

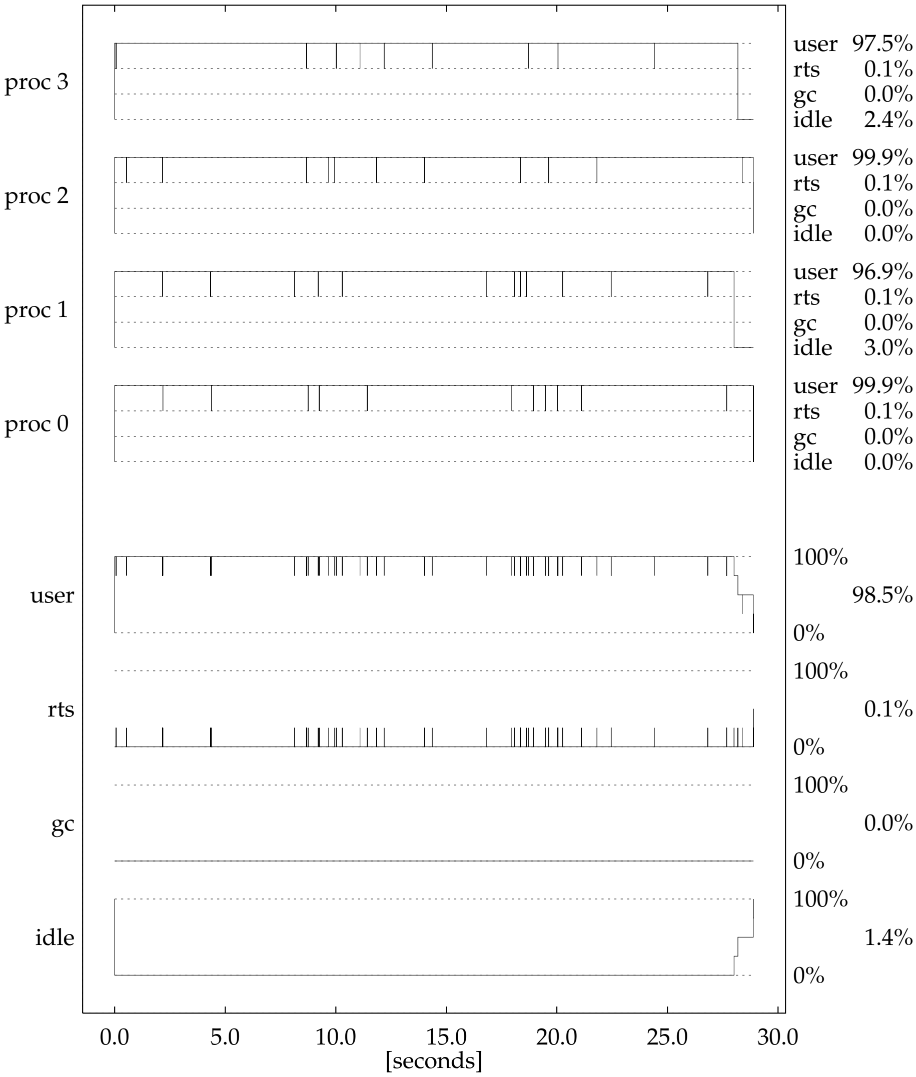


Figure 6.6: Execution profile of QUEENS: activity vs. time.

sparked; the application suffers slightly from load imbalance because of tasks with unequal lengths.

Note that none of the processor activity graphs shows any garbage collection taking place. This is caused by the nature of the QUEENS program in combination with the 1 ms sampling rate of the performance monitor. By default the garbage collector is called upon termination of each task to reclaim its garbage, but in case of QUEENS the collector is finished in a few instructions since the result value is a plain integer: no live data has to be copied to to-space at all. Therefore the state change from **rts** to **gc** and back is not detected by the performance monitor, hence, no spikes in the processor activity graphs of Figure 6.6.

The system activity graphs of an execution profile (e.g., the four bottom plots in Figure 6.6) show the summed totals of all processors for each activity. For example, the plot marked “idle” shows the number of idle processors during the execution; near the end of the QUEENS program the efficiency of the system slightly decreases since processors 1 and 3 are running idle. The “gc” plot does not show any activity, but for larger applications the plot alternates between zero and 25% (one processor) because of the single to-space.

The percentages listed at the right of each plot in the execution profile, give the (average) fraction of execution time spent in the corresponding state. For example, processor 0 has been busy executing user code for 99.9%, but on average the whole system has spent 98.5% of its time in user code. The QUEENS program behaves nicely since only 1.4% of the total execution time is wasted to processors running idle.

6.5.2 Wang

The execution profile of WANG is shown in Figure 6.7. Wang’s method of solving a tri-diagonal system of linear equations consists of two elimination phases. This can be observed in the execution profile: after 9 seconds processor 0 runs idle since all tasks of the first phase have been allocated for execution. When all processors have finished their last task of phase one, the root processor prepares the computations for the next phase. This takes little time, so all processors immediately start processing Phase 2 tasks to finish the application. The strict synchronisation halfway is clearly visible in the system plot for **idle** time (see bottom plot in Figure 6.7).

In each parallel phase the algorithm splits the problem into twelve independent components, which are scheduled as three consecutive tasks on each processor. When a tasks is finished the RTS compacts the result by running the garbage collector, as can be seen by the dips in the processor activity plots in Figure 6.7. Each task is of similar length, so after the first four tasks have been reduced to normal form, their processors have to compete for access to the single to-space in order to run the garbage collector. This shows up in

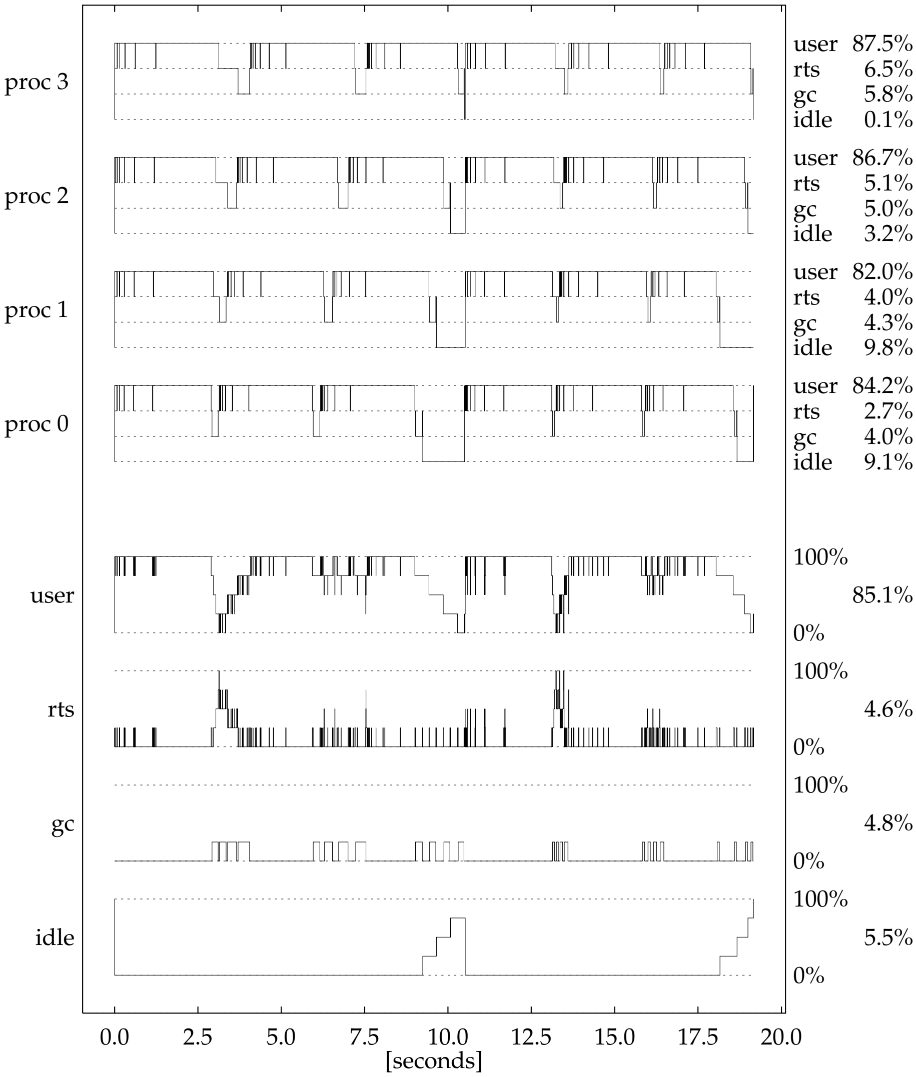


Figure 6.7: Execution profile of WANG: activity vs. time.

the execution profile: the system "gc" plot shows four consecutive garbage collects and the processor-activity plots show waiting in the state **rts**. From the execution profile, it can be determined that processor 0 collects first, followed by 1, 2, and 3 respectively. Note that the time to perform a garbage collect increases for each of the processors because of contention on the memory bus.

When processor 0 collects its garbage all others are waiting and do not use the memory bus; when processor 3 collects garbage all others have started the execution of a new task generating traffic on the memory bus.

After the evaluation of the second set of four tasks, the processors do not compete again for the garbage collector because the first conflict has shifted them out of phase. Processor 3 still needs more time to collect its garbage than processor 0, see the system "gc" plot, even though now the three other processors perform graph reduction in both cases. This small surprise is most likely caused by the context switch that occurs when starting to reduce a new task: the new task generates a large number of so called cold start misses to fetch the working set into the cache causing additional traffic on the memory bus in comparison to the effects of the stationary misses of the finishing tasks processor 0 competes with few stationary misses of the graph reducers running in parallel, while processor 3 competes with the cold start misses of graph reducers "loading" the working set of their new tasks into the cache. Thus processor 0 incurs fewer access collisions on the memory bus, hence it takes less time to collect garbage than processor 3. The third round of garbage collects takes roughly equal time at all four processors because of low traffic on the memory bus; processors are either running idle or performing graph reduction with a working set that completely resides in the cache.

The garbage collection behaviour described above is also present in the second phase of the Wang algorithm.

6.5.3 Wave

The WAVE program is a typical scientific application that models the behaviour of some physical system by simulation. It uses the finite element computation method to determine the water heights in a model of the North Sea. In a number of consecutive time steps the new state is computed based on the interactions with neighbouring grid points as recorded in temporary matrices holding the physical properties of interest. This iterative behaviour results in a chain task structure as displayed in Figure 6.8.

The WAVE application shown in the execution profile of Figure 6.9 iterates three time steps, computing three state matrices at each iteration, which results in 9 computation phases of 64 parallel tasks each. These 9 phases can be identified in the execution profile by looking at the system idle state, which shows 9 dips indicating all processors are busy. During each phase a considerable number of short garbage collections takes place, except for one particular

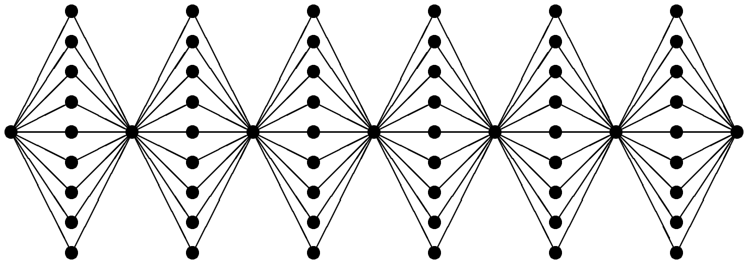


Figure 6.8: Chain task structure of WAVE.

collection halfway through the execution of WAVE that takes 1.3 seconds real time and leaves 3 processors running idle. Another waste of computing cycles is caused by the global synchronisation after each parallel phase. The effective parallelism of WAVE is restricted by the sequential parts in the algorithm that limit the maximum speed-up. The efficiency, however, can be raised by increasing the problem size.

The global garbage collection on processor 2 is performed on behalf of the sixth incarnation of the root task just before sparking the parallel tasks of Phase 7. After each parallel phase the root task continues processing with the new state matrices as computed by the child processes, while the previous state matrices have become obsolete. The accumulating garbage of "dead" state matrices can only be reclaimed when the root task is active; during each parallel phase the root task may not be collected locally since active children refer to matrices in the heap of the (suspended) root task. Therefore the RTS employs the heuristic that whenever the root task executes a sandwich and occupies more than 75% of the total heap space, the root has to reclaim garbage before sparking any new tasks. In case of the WAVE program, more than 800 Kbytes of live data (three 256×256 matrices with floating point numbers) have to be processed by the copying garbage collector, which takes 1.3 seconds. To reduce the garbage collection time, a number of approaches are possible. We discuss three of these alternatives.

First, the moment at which the root task invokes the garbage collector can be controlled by specifying a different threshold value for the default of 75% heap occupancy. Changing this limit, however, does not really help for WAVE since the size of the live data between each phase is large and constant. The WAVE program represents an important class of scientific applications, hence, additional effort is needed to decrease the impact of this global garbage collection bottleneck.

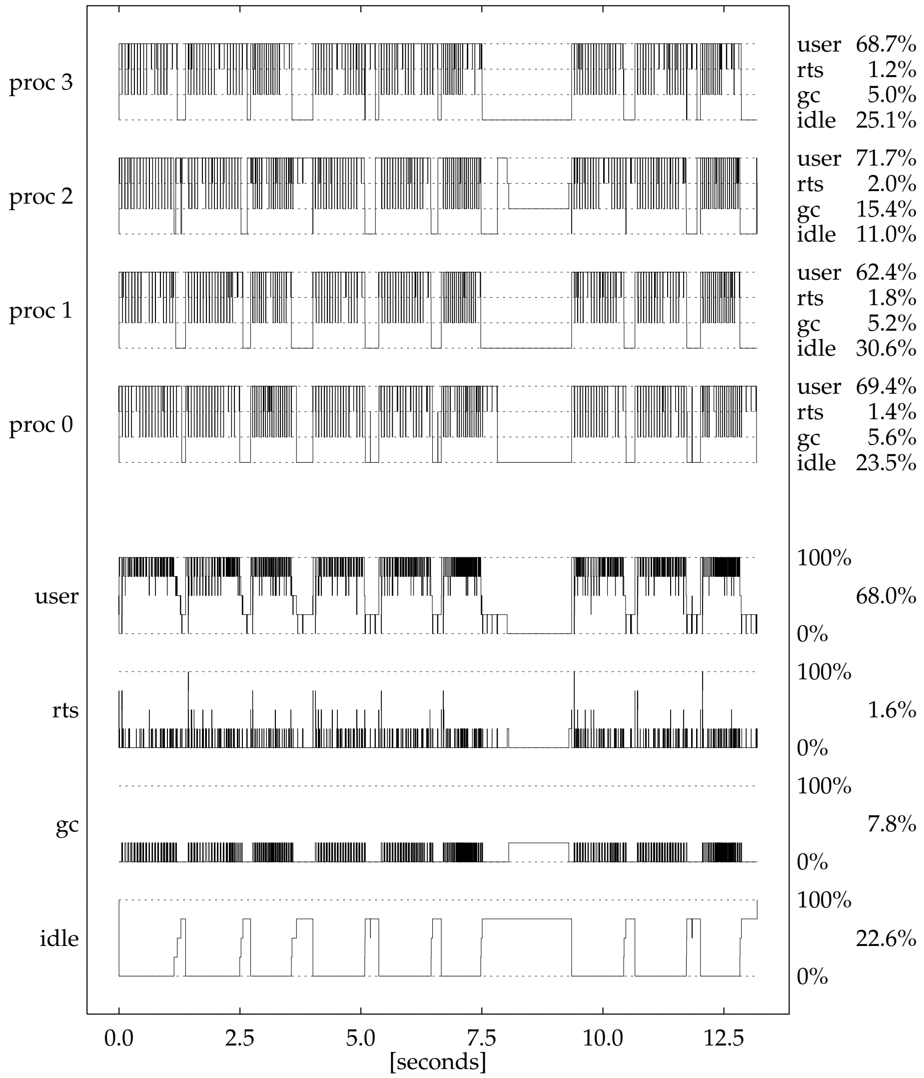


Figure 6.9: Execution profile of WAVE: activity vs. time.

Second, a straightforward possibility to decrease collection time is to use a garbage collector that runs in parallel on all processors (see Section 4.4). The additional overhead of synchronising parallel collection and bus contention, however, severely limits the reduction in garbage collection time; the usage of locks is expensive (WYBERT performs better than spark-and-wait with locks, see Section 6.1) and garbage collection slows down a factor of two because of bus contention (see WANG's **gc** profile plot in Figure 6.7). Therefore the (estimated) reduction in garbage collection time is too small to remove the global bottleneck.

Third, in the particular case of iterative methods, the programmer "knows" which data (i.e. state matrix) becomes obsolete in the following cycle. Thus, instead of reclaiming the unused space with an ordinary garbage collector, the matrix can be re-used immediately in the next iteration as a place holder for the result matrix. This approach known as *update analysis* has recently been used with success for first order functional languages [Cann92], and efforts have been undertaken to incorporate this in compilers for lazy higher-order languages as well [Bloss89].

6.5.4 Performance characteristics

This section presents the overall performance characteristics of the benchmark programs. The overhead of squeezing arguments and task handling has been measured on a single processor. Only WANG (5%) and 15-PUZZLE (18%) show increased execution times when the sandwich annotation is used; the other coarse-grain divide-and-conquer applications do not suffer any transformation loss. Unlike the experience with the SIS-simulator, none of the applications caused FRATS to get lost in a non-terminating reduction sequence. This is a consequence of the FAST/FCG compiler that does not perform fully lazy lambda lifting, hence, curried functions like '`map square`' can not be reduced and do not expand into infinite chains as in case of the SIS interpreter.

The example execution profiles of the QUEENS, WANG, and WAVE programs have shown that the monitoring tool as implemented in the prototype machine provides valuable insight into which application property determines performance: load imbalance in case of QUEENS, global synchronisation in case of WANG and WAVE. To obtain accurate measurements and study realistic application behaviour, the input parameters of the benchmark programs are set to large values. The resulting execution profiles have been summarised in Table 6.8 as a cost break-down of system activity: the average **idle**, **gc**, **rts**, and **user** time are reported for each application.

The column labeled seconds in Table 6.8 gives the seconds real time needed to execute the application on the four node *HYPERmodule* under a ToS scheduler with local task pools, the Virtual Allocation Scheme (VAS) for storage management, and local garbage collection. The remaining columns

	runtime [sec]	speed-up	idle [%]	gc [%]	rts [%]	user [%]
NFIB	55	4.0	0.5	0.0	0.0	99.5
QUEENS	160	3.9	1.2	0.0	0.2	98.6
DET	71	3.2	0.8	0.0	6.8	92.4
WANG	19	2.4	6.3	5.0	8.4	80.3
15-PUZZLE	54	2.7	6.7	1.5	17.1	74.7
COMP-LAB	28	2.6	24.3	4.2	4.1	67.4
WAVE	133	2.4	26.4	8.7	3.4	61.5

Table 6.8: Execution cost break-down for benchmark applications (large).

show in percentages where an application has spent its time as measured by the profiler.

The test NFIB program performs very good: it reaches an efficiency of over 99% on the four node machine. It consumes hardly any heap space and does not suffer from load imbalance. It is the "perfect" application to demonstrate system performance. The DET program is another test program since it does not spend time in garbage collection just like NFIB and QUEENS. In contrast to the others, DET claims a considerable number of heap cells (7% rts time) but the resident graph size is small so each garbage collect takes no time. The overall efficiency is good since 93% of the execution time is spent in user code.

Of the remaining serious applications, which do show garbage collection activity, the WANG program performs best: 80% efficiency on four processors. The execution cost break-down shows similar ratios as in the example execution profile on page 169. The **idle** time is caused by the synchronisations halfway and at the end of the program, while the **rts** time is partly spent in spinning on the lock of the single to-space. The 15-PUZZLE spent twice as much time in the runtime support system (17%) because it generates many fine-grain tasks. These small tasks incur the rather large initialisation overhead by the RTS causing the efficiency to drop to 75% (i.e. a speed-up of 3 on the four processor prototype machine).

Both the COMP-LAB and WAVE program suffer from a large fraction of idle time (25%). In case of COMP-LAB this is caused by a single synchronisation phase where individual join tasks, which combine the results of two child tasks, compute for a relatively long time. In case of WAVE, the iterative nature causes a sequence of global synchronisations leading to a severe loss of efficiency. As remarked before, a large fraction of the sequential thread of execution is taken by the global garbage collects needed to reclaim "dead" state of previous

	# processors		
	2	3	4
NFIB	1.00	1.00	1.00
QUEENS	1.00	1.00	1.00
DET	0.96	0.92	0.87
WANG	0.94	0.85	0.75
15-PUZZLE	0.97	0.93	0.89
COMP-LAB	1.00	0.98	0.96
WAVE	0.99	0.99	0.98

Table 6.9: Performance degradation caused by bus contention.

iterations. A special purpose collector can reduce this overhead considerably, and decrease the idle time to something like 15% (extrapolation of the WAVE execution profile).

There are several causes for the non-linear speed-ups reported in Table 6.8. First, some applications suffer from load-imbalance and sequential execution as can be inferred from the column that gives the percentage of **idle** time. Second, applications can spend a significant amount of time in blocking on semaphores (state **rts**), for example, to acquire the shared to-space. Third, memory bound applications suffer from contention on the bus to shared memory. This performance degradation caused contention has not been measured separately, and is included as part of the reported user time. To estimate the contention effects, Table 6.9 lists the ratio of user time fraction on a single processor over the fraction user time measured for multiple processors. These ratios show that WANG looses 25% performance on the four processor *HYPER-module* due to bus contention. The slight contention loss for COMP-LAB and WAVE indicates that their low speed-ups are caused completely by processors running idle.

The profile results of the seven benchmark programs with large input parameters show that the divide-and-conquer paradigm is suitable for efficient implementation and parallel execution of lazy functional programming languages on shared memory multiprocessors. Multi pass applications like COMP-LAB and WAVE, however, perform poorly because of the functional framework that forces a global synchronisation to pass information between "processes with state".

Chapter 7

Conclusions

The ever increasing demand for computing power is the driving force for the development of parallel processing. Building large parallel machines is relatively easy as can be seen from commercially available systems like the CM-5 and MasPar-2 which include thousands of processors. Programming parallel computers, however, is far more difficult and is an important field of research. The grand challenge for many computer scientists is to develop a suitable high-level programming environment that hides the low-level details of parallelism from the ordinary user.

Many different (parallel) programming languages have been developed for writing applications to run on parallel machines, but most language designs concentrate on raw application performance rather than ease-of-use. Functional programming languages contain a number of key properties that support general purpose parallel programming:

- Lazy evaluation and higher-order functions provide a high-level of abstraction to master software complexity.
- The referential transparency of functional programs provides simple semantics, so programs are easy to reason about and applicable to optimising transformations.
- Functional programming languages naturally support the shared-memory view of parallel programming. The user does not have to explicitly send messages to remote processors, but can communicate through (logically) shared data objects. Parallelism is obtained by annotating the program to indicate which expressions are suitable candidates for parallel execution.

A disadvantage of functional programming languages is the execution speed that is significantly lower than that of traditional imperative programming languages. The FAST/FCG compiler used in the WYBERT prototype implementation, whose code-generator has been described in detail in Chapter 5, generates state-of-the-art code as was shown by an extensive comparison with other lazy functional language compilers. Considerable effort has be put in the development of the FAST/FCG compiler since the user needs high absolute performance, not merely good speed-ups; parallelising overhead is rather easy in comparison to getting speed-up out of an efficient implementation.

The parallel implementation of a lazy functional language is not as straightforward as it seems at first sight. The belief that "the lack of side effects allows for easy parallel implementations" does not hold in general; even though the programmer can not (ab)use assignments in a functional language as in imperative languages, the underlying computational model of graph reduction heavily depends on updates of delayed computations to maintain sharing. Lazy evaluation without updates (i.e. string reduction) is hopelessly inefficient. To guarantee correctness in face of parallel access, the shared updatable redexes are usually protected by locks to enforce mutual exclusion. For certain classes of applications, however, the referential transparency can be exploited for parallel programming by automatic transformation. For example, applications specified as a parallel synchronous network can be automatically transformed into efficient divide-and-conquer programs, see [Vree92].

The WYBERT implementation discussed in this book uses a different method to handle updatable redexes present in parallel graph reduction systems on shared-memory multiprocessors. Instead of protecting shared redexes, the FRATS reduction strategy avoids their existence by eagerly evaluating all shared data between tasks before sparking them for parallel execution. This approach is feasible because WYBERT is based on an explicit divide-and-conquer annotation to express parallelism, so the compiler and runtime support are in control of task creation and synchronisation. An important advantage of WYBERT is that the banishment of shared redexes obviates the need for locks on application nodes.

The comparison between WYBERT and the spark-and-wait model on the *HYPERmodule* shared-memory multiprocessor presented in Section 6.1 shows that the avoidance of locking is beneficial for the divide-and-conquer benchmark applications. The advantage is not only gained by omitting several instructions when invoking a delayed computation stored in the heap, but primarily by reducing the number of transactions on the memory bus. This pays off in case of multiple processors since then the number of wasted cache cycles due to bus contention and snooping each others locking actions is significantly reduced. In case of the WAVE program WYBERT runs twice as fast as the spark-and-wait implementation that uses the *xmem* instruction of the MC88100 processor for locking application nodes. The benefit will be even larger for shared-memory multiprocessors with more than four processors.

A disadvantage of squeezing task arguments to banish shared redexes is that the deviation of the standard lazy evaluation mechanism might lead to superfluous or even non-terminating computations. Experience with the benchmark applications, however, has shown that the eager semantics of the squeeze does not raise problems in practice. The set of program transformations given in Section 4.2 has only been used for the fully lazy SIS simulator; no transformations were necessary for the compilation based experiments in Chapter 6. The experimental results measured on the four processor *HYPERmodule* show that the squeeze overhead pays off in performance: WYBERT without locking of application nodes is 25% - 50% faster than spark-and-wait with locking for the realistic benchmark applications.

The regular parallelism of the sandwich divide-and-conquer skeleton under the FRATS reduction strategy allows for two additional optimisations in the runtime support system of WYBERT: 1) argument stacks of tasks can be allocated on a single stack per processor instead of a stack per task, 2) garbage can be collected per individual leaf task without synchronisation of other tasks or processors. Both memory management optimisations are not possible in case of the general spark-and-wait model.

The optimal depth first traversal of the divide-and-conquer task tree can be mapped efficiently on a stack-based context switching mechanism. All tasks assigned to a specific processor share one reduction stack, the processor stack, as a stack of stacks. At start up, a task sets its private stack pointer to the current top of the processor stack. If the task executes a sandwich and needs to block to await the results of its children, the task leaves its local state on the processor stack, and the next fresh task starts to allocate its stack on top of the blocked task, etc. When a blocked task has received the results of all its children, it becomes executable again, but that task may only resume execution after all tasks on top of it have finished, otherwise it could overwrite the state of other tasks. Theoretically, this Top-of-Stack (ToS) scheduling constraint can lead to a severe loss parallelism as shown in Section 4.3, but the experimental results in Chapter 6 show that the ToS constraint does not lead to scheduling anomalies in practice.

In contrast to ToS, scheduling policies used in other parallel implementations of functional languages as discussed in 3.1.3 either give up stack based reduction or have a rather large context switch time. The $<\nu,G>$, HDG, and PAM implementations allocate their reduction stack as a linked list of stack frames in the heap. Besides the additional overhead of heap allocation and link manipulation, it destroys the strong cache locality of stack accesses [Langendoen92a]. The GRIP and PABC implementations allocate a fixed-length stack for each new task increasing the initialisation overhead, and forcing stack checks to enlarge the stack on overflow. The advantage of WYBERT's ToS scheduler is the combination of stack based reduction with fast context switching.

The second storage management optimisation is the local garbage collection of leaf tasks. Instead of synchronising all reducers in the multiprocessor to join in a global garbage collect, the WYBERT system assigns a private heap to each task and reclaims garbage per task independently of other tasks and processors. This approach is possible because the FRATS reduction strategy guarantees that no shared updatable redexes exist, hence, no task can obtain a reference into the private heap of another active task. When a task terminates, the private heap containing the result value is joined to the heap of the parent task. In order to efficiently collect such scattered parent heaps with an ordinary two-space copying garbage collector, WYBERT employs a memory management strategy that allocates (chunks of) private heaps such that the heap of a task never interleaves with the heap of its ancestors. The Virtual Allocation Scheme (VAS) uses the hardware support for virtual memory of the MC88200 MMU/cache chip to prevent interleaving by allocating a private heap in the first available virtual address range above the task's parent. The experiments in Chapter 6 have shown that the VAS policy, in accordance with the results of the MiG simulator in Section 4.4, can handle large application without running out of virtual address space. Furthermore VAS is reasonably efficient in comparison to the simpler BAS policy that can only handle small programs: just 8% slower in the worst case

The advantage of local garbage collection is that by limiting the maximum task size the amount of memory reserved as to-space can be decreased. By default WYBERT reserves $\frac{1}{5}$ of the available heap memory as a time-shared to-space, leaving $\frac{4}{5}$ for the graph reducers. In contrast, a global garbage collector needs $\frac{1}{2}$ the memory for garbage collection. In return for the additional heap space, which reduces the number of garbage collects, tasks under WYBERT occasionally have to wait for another task using the single to-space. Experiments have shown that such waiting occurs for perfect divide-and-conquer applications that generate sub problems of equal size like WANG and WAVE. Increasing the number of to-spaces, however, does not boost performance at all because the reduction of available heap space forces additional garbage collects. In case of WAVE, increasing to two to-spaces gives 3% better overall performance, but with four to-spaces performance *drops* with 38%.

The detailed performance measurements of benchmark applications with large parameter values, as reported in Section 6.5.4, show a considerable difference in performance: efficiency on the four processor *HYPERmodule* ranges from 62% for WAVE to 99% for QUEENS. The low efficiency of WAVE is caused by the iterative nature of the application of this typical scientific simulation model. After several iteration steps, the amount of accumulated garbage in the root has become so large that a (global) garbage collection is necessary. Because of the large state matrix, this takes considerable time and stretches the bottleneck of sequential execution. The divide-and-conquer paradigm, which forces global synchronisations to communicate information between "processes", in combination with local garbage collection per task results in rather poor performance for iterative algorithms. Switching to global garbage collection does not solve the problem because of reducing the amount of available heap space (38% decrease in performance when using four to-spaces), but switching to spark-and-wait is no solution either because of the additional overhead of locking (50% decraese in performance). Therefore additional research is needed to find a practical solution.

The tools for parallel programming used and developed during the research described in this book have proven to be of great value. In particular the MiG simulator offers a good platform for debugging runtime support software in a deterministic environment; unlike with real hardware, bugs can be reproduced by simply running the program again with the same input parameters. The monitoring tool that samples the multiprocessor state is another useful tool, which has been heavily used to explain the runtime behaviour of the application benchmark. Both the MiG simulator and monitoring tool do not depend on specific functional language properties and can be used for other parallel programming language implementations as well.

The performance measurements in Chapter 6 have shown that lock-free graph reduction in combination with an efficient context switching mechanism (ToS) and local storage management (VAS) provides a highly efficient implementation for the parallel execution on shared-memory multiprocessors of divide-and-conquer applications written in a lazy functional language.

Bibliography

[Abramsky87] S. Abramsky and C. Hankin, editors. *Abstract interpretation of declarative languages.* Ellis Horwood, Chichester, England, 1987.

[Achten91] P. Achten. Annotations for load distribution. In H. W. Glaser and P. H. Hartel, editors, *3rd Implementation of functional languages on parallel architectures*, pages 247–264, Southampton, England. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, 1991.

[Aho86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, techniques, and tools.* Addison Wesley, Reading, Massachusetts, 1986.

[Appel88] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Programming language design and implementation*, pages 11–20, Atlanta, Georgia. ACM SIGPLAN notices,23(7), 1988.

[Appel89] A. W. Appel. Runtime tags aren't necessary. *Lisp and symbolic computation*, 2(2):153–162, 1989.

[Augustsson84] L. Augustsson. A compiler for lazy ML. In *Lisp and functional programming*, pages 218–229, Austin, Texas. ACM, 1984.

[Augustsson89a] L. Augustsson and T. Johnsson. The Chalmers lazy-ML compiler. *The computer journal*, 32(2):127–141, 1989.

[Augustsson89b] L. Augustsson and T. Johnsson. Parallel graph reduction with the $\langle\nu$,G$\rangle$-machine. In J. Stoy, editor, *4th Functional programming languages and computer architecture*, pages 202–213, London, England. ACM, 1989.

[Augustsson90] L. Augustsson and T. Johnsson. Lazy ML user's manual. Programming methodology group report, Dept. of Comp. Sci, Chalmers Univ. of Technology, Göteborg, Sweden, 1990.

[Bal90] H. E. Bal. *Programming Distributed Systems.* Silicon Press, Summit NJ, USA, 1990.

[Barendregt84] H. P. Barendregt. *The lambda calculus, its syntax and semantics.* North Holland, Amsterdam, The Netherlands, 1984.

[Barendregt87] H. P. Barendregt, M. C. J. D. van Eekelen, P. H. Hartel, L. O. Hertzberger, M. J. Plasmeijer, and W. G. Vree. The Dutch parallel reduction machine project. *Future generation computer systems*, 3(4):261–270, 1987.

[Barendregt92] H. P. Barendregt, M. Beemster, P. H. Hartel, L. O. Hertzberger, R. F. H. Hofman, K. G. Langendoen, L. L. Li, R. Milikowski, J. C. Mulder, and W. G. Vree. Programming clustered parallel reduction machines. Technical report CS-92-05, Dept. of Comp. Sys, Univ. of Amsterdam, 1992.

[Bevan87] D. I. Bevan. Distributed garbage collection using reference counting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *1st Parallel architectures and languages Europe (PARLE), LNCS 258/259*, pages 176–187, Eindhoven, The Netherlands. Springer-Verlag, 1987.

[Bird84] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta informatica*, 21(3):239–250, 1984.

[Bird88] R. S. Bird and P. L. Wadler. *Introduction to functional programming.* Prentice Hall, New York, 1988.

[Birrell84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM transactions on computer systems*, 2(1):39–59, 1984.

[Bloss89] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In J. Stoy, editor, *4th Functional programming languages and computer architecture*, pages 26–38, London, England. ACM, 1989.

[Bokhari92] S. H. Bokhari and H. Berryman. Complete exchange on a circuit-switched mesh. In *Proceedings of SHPCC '92*, pages 300–306, Williamsburg, VA. IEEE, 1992.

[Brownbridge85] D. R. Brownbridge. Cyclic reference counting for combinator machines. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 273–288, Nancy, France. Springer-Verlag, 1985.

[Burn91] G. L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman publishing, London, UK, 1991.

[Burton85] F. W. Burton. Speculative computation, parallelism and functional programming. *IEEE transactions on computers*, C-34(12):1190–1193, 1985.

[Cann92] D. C. Cann. Retire FORTRAN? a debate rekindled. *Communications ACM*, 35(8):81–89, 1992.

[Carriero89] N. Carriero and D. Gelernter. Linda in context. *Communications ACM*, 32(4):444–458, 1989.

[Cheney70] C. J. Cheney. A non-recursive list compacting algorithm. *Communications ACM*, 13(11):677–678, 1970.

[Cohen81] J. Cohen. Garbage collection of linked structures. *ACM computing surveys*, 13(3):341–367, 1981.

[Cox92] S. Cox, S.-Y. Huang, P. H. J. Kelly, J. J. Liu, and F. Taylor. An implementation of static functional process networks. In D. Etiemble and J.-C. Syre, editors, *4th Parallel architectures and languages Europe (PARLE), LNCS 605*, pages 497–512, Paris, France. Springer-Verlag, 1992.

[Crammond88] J. Crammond. A garbage collection algorithm for shared memory parallel processors. *Journal parallel programming*, 17(6):497–522, 1988.

[Darlington81] J. Darlington and M. Reeve. ALICE: A multiple-processor reduction machine for the parallel evaluation of applicative languages. In Arvind, editor, *1st Functional programming languages and computer architecture*, pages 65–76, Wentworth-by-the-Sea, Portsmouth, New Hampshire. ACM, 1981.

[Darlington91] J. Darlington, A. J. Field, P. G. Harrison, D. Harper, G. K. Jouret, P. H. J. Kelly, K. M. Sephton, and D. W. Sharp. Structured parallel functional programming. In H. W. Glaser and P. H. Hartel, editors, *3rd*

Implementation of functional languages on parallel architectures, pages 31–51, Southampton, England. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, 1991.

[Davy92] J. R. Davy. *Using divide and conquer for parallel geometric evaluation.* PhD thesis, School of Computer Studies, Univ. of Leeds, England, 1992.

[Derbyshire90] M. H. Derbyshire. Mark scan garbage collection on a distributed architecture. *Lisp and symbolic computation*, 3(2):135–170, 1990.

[Eager89] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Computers*, 38(3):408–423, 1989.

[Ein-Dor85] P. Ein-Dor. Grosch's law revisited: CPU power and the cost of computation. *Communications ACM*, 28(2):142–151, 1985.

[Embrechts90] H. Embrechts, D. Roose, and P. Wambacq. Component labelling on a MIMD multiprocessor. prepublished report, Dept. of Comp. Sci., Katholieke Universiteit Leuven, Belgium, 1990.

[Field88] A. J. Field and P. G. Harrison. *Functional programming.* Addison Wesley, Reading, Massachusetts, 1988.

[Fleming86] P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications ACM*, 29(3):218–221, 1986.

[Flynn72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, C-21(9):948–960, 1972.

[Gabriel84] R. P. Gabriel and J. McCarthy. Queue-based multi-processing Lisp. In *Lisp and functional programming*, pages 25–44, Austin, Texas. ACM, 1984.

[George89] L. George. An abstract machine for parallel graph reduction. In J. Stoy, editor, *4th Functional programming languages and computer architecture*, pages 214–229, London, England. ACM, 1989.

[Glas92] J. Glas. The parallelization of branch and bound algorithms in a functional programming language. Master's thesis, Dept. of Comp. Sys, Univ. of Amsterdam, 1992.

[Goldberg88a] B. Goldberg. Multiprocessor execution of functional programs. *International Journal of Parallel Programming*, 17(5):425–473, 1988.

[Goldberg88b] B. F. Goldberg. Buckwheat: Graph reduction on a shared memory multiprocessor. In *Lisp and functional programming*, pages 40–51, Snowbird, Utah. ACM, 1988.

[Goldberg88c] B. F. Goldberg. *Multiprocessor execution of functional programs.* PhD thesis, Dept. of Comp. Sci, Yale Univ., 1988.

[Gottlieb83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomupter - designing an MIMD shared memory parallel computer. *IEEE transactions on computers*, C-32(2):175–189, 1983.

[Graham69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal applied mathematics*, 17(2):416–429, 1969.

[van Groningen92] J. H. G. van Groningen. Some implementation aspects of Concurrent Clean on distributed memory architectures. In H. Kuchen and R. Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, pages 403–414, Aachen, Germany. Aachener Informatik-Berichte 92-19, RWTH Aachen, Fachgruppe Informatik, 1992.

[Gurd85] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications ACM*, 28(1):34–52, 1985.

[Halstead Jr84] R. H. Halstead Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Lisp and functional programming*, pages 9–17, Austin, Texas. ACM, 1984.

[Hammond91] K. Hammond and S. L. Peyton Jones. Profiling strategies on the GRIP parallel reducer. Internal report 10, Dept. of Comp. Sci, Univ. of Glasgow, Scotland, 1991.

[Hartel88a] P. H. Hartel. *Performance analysis of storage management in combinator graph reduction.* PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, 1988.

[Hartel88b] P. H. Hartel and A. H. Veen. Statistics on graph reduction of SASL programs. *Software—practice and experience*, 18(3):239–253, 1988.

[Hartel89] P. H. Hartel, M. H. M. Smid, L. Torenvliet, and W. G. Vree. A parallel functional implementation of range queries. In P. G. M. Apers, D. Bosman, and J. van Leeuwen, editors, *Computing science in the The Netherlands*, pages 173–189, Utrecht, The Netherlands. CWI, 1989.

[Hartel90] P. H. Hartel. A comparison of three garbage collection algorithms. *Structured programming*, 11(3):117–127, 1990.

[Hartel91a] P. H. Hartel, H. W. Glaser, and J. M. Wild. Compilation of functional languages using flow graph analysis. Technical report CSTR 91-03, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, 1991.

[Hartel91b] P. H. Hartel, H. W. Glaser, and J. M. Wild. On the benefits of different analyses in the compilation of functional languages. In H. W. Glaser and P. H. Hartel, editors, *3rd Implementation of functional languages on parallel architectures*, pages 123–145, Southampton, England. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, 1991.

[Hartel92] P. H. Hartel and W. G. Vree. Arrays in a lazy functional language – a case study: the fast Fourier transform. In G. Hains and L. M. R. Mullin, editors, *2nd Arrays, functional languages, and parallel systems (ATABLE)*, pages 52–66. Publication 841, Dept. d'informatique et de recherche opérationelle, Univ. de Montréal, Canada, 1992.

[Hartel93] P. H. Hartel and K. G. Langendoen. Benchmarking implementations of lazy functional languages. In *6th Functional programming languages and computer architecture*, pages 341–349, Copenhagen, Denmark. ACM, 1993.

[Hellberg92] S. Hellberg, I. Glendinning, and P. Shallow. Tools for parallel high performance systems - comparative evaluation. Technical report SNARC 92-01, Univ. of Southampton, UK, 1992.

[Hertzberger89] L. O. Hertzberger and W. G. Vree. A coarse grain parallel architecture for functional languages. In *PARLE 1989, LNCS 365/366*, pages 269–285, Eindhoven, The Netherlands. Springer-Verlag, 1989.

[Hofman92a] R. F. H. Hofman and W. G. Vree. Distributed hierarchical scheduling with explicit grain size control. *Future Generation Computer Systems*, 8:111–119, 1992.

[Hofman92b] R. F. H. Hofman, K. G. Langendoen, and W. G. Vree. Scheduling consequences of keeping parents at home. In *Parallel and Distributed Systems (ICPADS)*, pages 580–588, HsinChu, Taiwan. National Tsing Hwa Univ., 1992.

[Hofman93] R. F. H. Hofman. *Scheduling and grain size control.* PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, 1993.

[Hudak86] P. Hudak and L. Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *13th Principles of programming languages*, pages 243–254, St. Petersburg Beach, Florida. ACM, 1986.

[Hudak89] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[Hudak92] P. Hudak, S. L. Peyton Jones, and P. L. W. (editors). Report on the programming language Haskell – a non-strict purely functional language, version 1.2. *ACM SIGPLAN notices*, 27(5):1–162, 1992.

[Hughes82] R. J. M. Hughes. Super combinators – a new implementation method for applicative languages. In *Lisp and functional programming*, pages 1–10, Pittsburg, Pennsylvania. ACM, 1982.

[Hughes83] R. J. M. Hughes. *The design and implementation of programming languages.* PhD thesis, Oxford Univ, England, 1983.

[Hughes89] R. J. M. Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[Hughes90] R. J. M. Hughes. Compile-time analysis of functional programs. In D. A. Turner, editor, *Research topics in functional programming*, pages 117–153, Reading, Massachusetts. Addison Wesley, 1990.

[Johnsson84] T. Johnsson. Efficient compilation of lazy evaluation. In *Compiler construction*, pages 58–69, Montréal, Canada. ACM SIGPLAN notices,19(6), 1984.

[Johnsson87] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *3rd Functional programming languages and computer architecture, LNCS 274*, pages 154–173, Portland, Oregon. Springer-Verlag, 1987.

[Kahn74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden. North Holland, 1974.

[Kelly89] P. H. J. Kelly. *Functional programming for loosely-coupled multiprocessors*. Pitman publishing, London, England, 1989.

[Kennaway83] J. R. Kennaway and M. R. Sleep. Novel architectures for declarative languages. *Software and microsystems*, 2(3):59–70, 1983.

[Kernighan78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1978.

[Kesseler91] M. Kesseler. Implementing the PABC machine on Transputers. In H. W. Glaser and P. H. Hartel, editors, *3rd Implementation of functional languages on parallel architectures*, pages 409–421, Southampton, England. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, 1991.

[Kesseler92] M. Kesseler. Communication issues regarding parallel functional graph rewriting. In H. Kuchen and R. Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, pages 417–435, Aachen, Germany. Aachener Informatik-Berichte 92-19, RWTH Aachen, Fachgruppe Informatik, 1992.

[Kingdon91] H. Kingdon, D. R. Lester, and G. L. Burn. The HDG-machine: a highly distributed graph-reducer for a transputer network. *The computer journal*, 34(4):290–301, 1991.

[Kluge83] W. E. Kluge. Cooperating reduction machines. *IEEE transactions on computers*, C-32(11):1002–1012, 1983.

[Kranz89] D. A. Kranz, R. H. Halstead, and E. Mohr. Mul-t: A high-performance parallel Lisp. In *Programming language design and implementation*, pages 81–90, Portland, Oregon. ACM SIGPLAN notices,24(7), 1989.

[Landin64] P. J. Landin. The mechanical evaluation of expressions. *The computer journal*, 6(4):308–320, 1964.

[Langendoen91a] K. G. Langendoen and W. G. Vree. Eight queens divided: An experience in parallel functional programming. In J. Darlington and

R. Dietrich, editors, *Declarative programming*, pages 101–115, Sasbachwalden, West Germany. Springer-Verlag, 1991.

[Langendoen91b] K. G. Langendoen and W. G. Vree. FRATS: a parallel reduction strategy for shared memory. In J. Maluszynski and M. Wirsing, editors, *3rd Programming language implementation and logic programming, LNCS 528*, pages 99–110, Passau, West Germany. Springer-Verlag, 1991.

[Langendoen92a] K. G. Langendoen and D. J. Agterkamp. Cache behaviour of lazy functional programs. In H. Kuchen and R. Loogen, editors, *4th Parallel implementation of functional languages*, pages 33–46, Aachen, Germany. Aachener Informatik-Berichte 92-19, RWTH Aachen, Fachgruppe Informatik, 1992.

[Langendoen92b] K. G. Langendoen, H. L. Muller, and W. G. Vree. Memory management for parallel tasks in shared memory. In Y. Bekkers and J. Cohen, editors, *Memory management (IWMM), LNCS 637*, pages 165–178, St. Malo, France. Springer-Verlag, 1992.

[Lester89a] D. R. Lester. An efficient distributed garbage collection algorithm. In E. Odijk, M. Rem, and J.-C. Syre, editors, *2nd Parallel architectures and languages Europe (PARLE), LNCS 365/366*, pages 207–223, Eindhoven, The Netherlands. Springer-Verlag, 1989.

[Lester89b] D. R. Lester. Stacklessness: compiling recursion for a distributed architecture. In J. Stoy, editor, *4th Functional programming languages and computer architecture*, pages 116–128, London, England. ACM, 1989.

[Li89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM transactions on computer systems*, 7(4):321–359, 1989.

[Liebermann83] H. Liebermann and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications ACM*, 26(6):419–429, 1983.

[Loogen89] R. Loogen, H. Kuchen, K. Indermark, and W. Damm. Distributed implementation of programmed graph reduction. In E. Odijk, M. Rem, and J.-C. Syre, editors, *2nd Parallel architectures and languages Europe (PARLE), LNCS 365/366*, pages 136–157, Eindhoven, The Netherlands. Springer-Verlag, 1989.

[Magó79] G. A. Magó. A network of microprocessors to execute reduction languages – part I. *Journal computer and information sciences*, 8(5):349–385, 1979.

[Maranget91] L. Maranget. GAML: a parallel implementation of lazy ML. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture, LNCS 523*, pages 102–123, Cambridge, Massachusetts. Springer-Verlag, 1991.

[McCarthy60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications ACM*, 3(4):184–195, 1960.

[Mohr91] E. Mohr, D. A. Kranz, and R. H. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE transactions on parallel and distributed systems*, 2(3):264–280, 1991.

[Muller93] H. L. Muller. *Simulating computer architectures.* PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, 1993.

[Nöcker91a] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *3rd Parallel architectures and languages Europe (PARLE), LNCS 505/506*, pages 202–220, Veldhoven, The Netherlands. Springer-Verlag, 1991.

[Nöcker91b] E. G. J. M. H. Nöcker, M. J. Plasmeijer, and S. Smetsers. The parallel ABC machine. In H. W. Glaser and P. H. Hartel, editors, *3rd Implementation of functional languages on parallel architectures*, pages 351–382, Southampton, England. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, 1991.

[Pehousek89] J. D. Pehousek and J. S. Weening. Low-cost process creation and a dynamic partitioning in Qlisp. In T. Ito and R. H. Halstead Jr, editors, *Parallel Lisp languages and systems, LNCS 441*, pages 182–199, Sendai, Japan. Springer-Verlag, 1989.

[Peyton Jones87a] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP – a high performance architecture for parallel graph reduction. In G. Kahn, editor, *3rd Functional programming languages and computer architecture, LNCS 274*, pages 98–112, Portland, Oregon. Springer-Verlag, 1987.

[Peyton Jones87b] S. L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[Peyton Jones89] S. L. Peyton Jones, C. Clack, and J. Salkild. High-performance parallel graph reduction. In E. Odijk, M. Rem, and J.-C. Syre, editors, *2nd Parallel architectures and languages Europe (PARLE), LNCS 365/366*, pages 193–206, Eindhoven, The Netherlands. Springer-Verlag, 1989.

[Peyton Jones92] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal functional programming*, 2(2):127–202, 1992.

[Pfister85] G. F. Pfister and V. A. Norton. Hot spot contention and combining in multistage interconnection networks. *IEEE transactions on computers*, C-34(10):943–948, 1985.

[Röjemo92] N. Röjemo. A concurrent generational garbage collector for a parallel graph reducer. In Y. Bekkers and J. Cohen, editors, *Memory management (IWMM), LNCS 637*, pages 440–453, St. Malo, France. Springer-Verlag, 1992.

[Rudalics86] M. Rudalics. Distributed copying garbage collection. In *Lisp and functional programming*, pages 364–372, Boston, Massachusetts. ACM, 1986.

[Ruggiero87] C. A. Ruggiero and J. Sargeant. Control of parallelism in the Manchester data flow machine. In G. Kahn, editor, *3rd Functional programming languages and computer architecture, LNCS 274*, pages 1–15, Portland, Oregon. Springer-Verlag, 1987.

[Schulte91] W. Schulte and W. Grieskamp. Generating efficient portable code for a strict applicative language. In J. Darlington and R. Dietrich, editors, *Declarative programming*, pages 239–252, Sasbachwalden, West Germany. Springer-Verlag, 1991.

[Smetsers93] J. E. W. Smetsers, E. Barendsen, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Guaranteeing destructive updatability through a type system with uniqueness information for graphs. Technical report 93-3, Dept. of Comp. Sci, Univ. of Nijmegen, The Netherlands, 1993.

[Smetsers91] S. Smetsers, E. G. J. M. H. Nöcker, J. van Groningen, and M. J. Plasmeijer. Generating efficient code for lazy functional languages. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture, LNCS 523*, pages 592–617, Cambridge, Massachusetts. Springer-Verlag, 1991.

[Stenström90] P. Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE computer*, 23(6):12–24, 1990.

[Stout87] Q. F. Stout. Supporting divide-and-conquer algorithms for image processing. *Journal parallel and distributed computing*, 4(1):95–115, 1987.

[Stoye84] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *Lisp and functional programming*, pages 159–166, Austin, Texas. ACM, 1984.

[Swan77] R. J. Swan, S. H. Fuller, and D. P. Siewiorek. Cm* – a modular, multimicroprocessor system. *Proceedings National Computer Conference*, pages 645–655, 1977.

[Sweazy86] P. Sweazy and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *13th IEEE/ACM symp. computer architecture*, pages 414–423. SIGARCH newsletter,14(2), 1986.

[Treleaven82] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins. Data-driven and demand-driven computer architecture. *ACM computing surveys*, 14(1):93–142, 1982.

[Turner79a] D. A. Turner. A new implementation technique for applicative languages. *Software—practice and experience*, 9(1):31–49, 1979.

[Turner79b] D. A. Turner. SASL language manual. Technical report, Computing Laboratory, Univ. of Kent at Canterbury, 1979.

[Turner81] D. A. Turner. The semantic elegance of applicative languages. In Arvind, editor, *1st Functional programming languages and computer architecture*, pages 85–92, Wentworth-by-the-Sea, Portsmouth, New Hampshire. ACM, 1981.

[Turner85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, Nancy, France. Springer-Verlag, 1985.

[Turner90] D. A. Turner. *Miranda system manual.* Research Software Ltd, 23 St Augustines Road, Canterbury, Kent CT1 1XP, England, 1990.

[Vegdahl84] S. R. Vegdahl. A survey of proposed architectures for executing functional languages. *IEEE transactions on computers*, C-33(12):1050–1071, 1984.

[Vrancken90] J. L. M. Vrancken and M. J. P. (ed.). Reflections on parallel functional languages. *Technical Report 90-16, Departement of Informatics, University of Nijmegen*, 1990.

[Vree89] W. G. Vree. *Design considerations for a parallel reduction machine.* PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, 1989.

[Vree90] W. G. Vree. Implementation of parallel graph reduction by explicit annotation and program transformation. In B. Rovan, editor, *Mathematical foundations of computer science, LNCS 452*, pages 135–151, Banská Bystrica, Czechoslovakia. Springer-Verlag, 1990.

[Vree92] W. G. Vree and P. H. Hartel. Fixed point computation for parallelism. Technical report CS-92-07, Dept. of Comp. Sys, Univ. of Amsterdam, 1992.

[Wadler90] P. L. Wadler. Comprehending monads. In *Lisp and functional programming*, pages 61–78, Nice, France. ACM, 1990.

[Wadsworth71] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus.* PhD thesis, Oxford Univ, England, 1971.

[Waite91] M. Waite, B. Giddings, and S. Lavington. Parallel associative combinator evaluation. In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *3rd Parallel architectures and languages Europe (PARLE), LNCS 505/506*, pages 331–348, Veldhoven, The Netherlands. Springer-Verlag, 1991.

[Wang81] H. H. Wang. A parallel method for tri-diagonal equations. *ACM transactions on mathematical software*, 7(2):170–183, 1981.

[Watson88] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. Flagship: A parallel architecture for declarative programming. In *15th IEEE/ACM symp. computer architecture*, pages 124–130, Honolulu, Hawaii. ACM SIGARCH newsletter,16(2), 1988.

[Watson86] P. Watson and I. Watson. Graph reduction in a parallel virtual memory environment. In J. F. Fasel and R. M. Keller, editors, *Graph reduction, LNCS 279*, pages 265–214, Santa Fé, New Mexico. Springer-Verlag, 1986.

[Watson87a] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *1st Parallel architectures and languages Europe (PARLE), LNCS 258/259*, pages 432–434, Eindhoven, The Netherlands. Springer-Verlag, 1987.

[Watson87b] P. Watson and I. Watson. Evaluation of functional programs on the Flagship machine. In G. Kahn, editor, *3rd Functional programming languages and computer architecture, LNCS 274*, pages 80–97, Portland, Oregon. Springer-Verlag, 1987.

Index