Profiling scheduling strategies on the GRIP parallel reducer

Kevin Hammond and Simon Peyton Jones *

August 23, 1991

Abstract

It is widely claimed that functional languages are particularly suitable for programming parallel computers. A claimed advantage is that the programmer is not burdened with details of task creation, placement, scheduling, and synchronisation, these decisions being taken by the system instead. Leaving aside the question of whether a pure functional language is expressive enough to encompass all the parallel algorithms we might wish to program, there remains the question of how effectively the compiler and run-time system map the program onto a real parallel system, a task usually carried out mostly by the programmer. This is the question we address in this paper.

We first introduce the system architecture of GRIP, a shared-memory parallel machine supporting an implementation of the functional language HASKELL. GRIP executes functional programs in parallel using compiled supercombinator graph reduction, a form of declarative rule system.

We then describe several strategies for run-time resource control which we have tried, presenting comprehensive measurements of their effectiveness. We are particularly concerned with strategies controlling task creation, in order to improve task granularity and minimise communication overheads. This is, so far as we know, one of the first attempts to make a systematic study of task-control strategies in a high-performance parallel functional-language system. GRIP's high absolute performance render these results credible for real applications.

1 Introduction

It has long been claimed that functional programming languages are particularly suitable for programming parallel computers.

We will not rehearse this argument here, except to identify its main basis: namely that, compared with parallel imperative languages, *much of the burden of resource allocation is shifted from the programmer to the system.* Since the parallelism is implicit, the programmer does not have to describe the details of thread creation, placement, scheduling, communication, and synchronisation[1].

*Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, UK. Tel: +44-41-339-8855 ext {5619,4500}. Email: {kh,simonpj}cs.glasgow.ac.uk

[1]Throughout this paper we use the term *thread* for a sequential activity which (potentially) runs concurrently with other threads.

If this claim is valid then the advantages will be considerable: it should be easier to write correct programs, and programs should be much more independent of the target architecture than their more explicit counterparts. To substantiate the claim, though, functional programmers need to address two main questions:

- Firstly, *are functional programming languages sufficiently expressive to encompass the parallel algorithms we would like to run*? Notice that we dismiss immediately the unrealistic expectation that an arbitrary functional program will speed up when run on a parallel machine. Rather, we approach a parallel machine with a parallel algorithm in mind: the question is whether we can readily express our thoughts in the medium of a functional programming language.

  In some cases, such as divide-and-conquer algorithms, the answer is plainly yes. In others, such as branch-and-bound algorithms, the picture is altogether murkier. This question has received some attention from researchers, many of whom have proposed extensions to the language to increase their expressiveness. Examples include the MIT dataflow work, which introduces I-structures [AE87]; Warren Burton's work on improving values for branch-and-bound algorithms [Bur89]; Hudak's "para-functional" programming style [Hud86]; Hughes and O'Donnell's proposals for parallel non-deterministic algorithms [HOD89]; and Roe's recent thesis [Roe91]. For the most part, this work is unproven in practice, and needs to be implemented and evaluated on real parallel systems.

- Secondly, *does the system make effective use of the resources of the parallel machine*? Mapping the requirements of a parallel program onto the resources of a multiprocessor is a very complex task. Issues of thread creation, placement and synchronisation consume a large proportion of the brain-power of parallel programmers. Worse, the effectiveness or otherwise of resource-control strategies makes a substantial impact on performance. Handing over control of these resources to the system is a two-edged weapon: it relieves the programmer of a difficult and error-prone task, but it risks losing a large performance margin if the system does a worse job than the programmer.

  If the performance loss is sufficiently small, or there is a performance gain, then a parallel functional system will be useful for all but the most stringent applications (expressiveness aside). If the performance loss is always substantial, then parallel functional programming will find few customers. Paged virtual memory is a useful analogy here: the performance loss from automatic paging (compared with explicit overlays) is so small, and the reduction in program complexity so great, that for most applications we barely consider the paging system at all. Only for the most demanding applications do we try to take more control over paging strategies.

Our research goals include both of these questions. In our view, they have not been much addressed so far because there have been few attempts to build a parallel functional-language system which delivers substantial *absolute* speedup over the best sequential implemenation of the same langauge, let alone of a FORTRAN program for the same problem. It is very difficult to persuade an application programmer to learn a totally new language, recode his application, only to be rewarded with a substantial loss of performance! Similarly, it is hard to make credible statements about the effectiveness with which a parallel functional program is mapped onto a parallel machine if there is a substantial slow-down factor to discount first.

Apart from our own work, the only other researchers who have reported a parallel functional-language system with this absolute-speed-up property are Johnsson and Augustsson, whose

$\langle\nu, G\rangle$-machine was the first to achieve this property.

In an attempt to bite this particular bullet, we have built

- A multiprocessor, GRIP (Graph Reduction in Parallel), designed to execute functional programs in parallel using graph reduction, a form of declarative rule system. GRIP's architecture is described further below (Section 3).
- A compiler for the HASKELL functional programming language [HW90]. This compiler is based on an earlier compiler for the LML language from Chalmers University [Joh84], with a new code generator organised around the Spineless Tagless G-machine [Pey89]. It generates native code for both Sun3 workstations and the GRIP multiprocessor.

GRIP runs parallel HASKELL programs with substantial absolute speedup over the same program running on a uniprocessor Sun with a comparable microprocessor [HP90]. We have now begun to make HASKELL and GRIP available to a number of colleagues in the UK who are using it to write parallel applications. This work is at an early stage as yet, but we are committed to "closing the loop", by using the experience of our users to guide our priorities in further development.

In this paper we make no attempt to discuss the issue of expressiveness. Instead, we focus mainly on two aspects of resource management: thread creation and thread scheduling.

We begin with an overview of the GRIP system, and how we perform compiled graph reduction. This is followed by a discussion of scheduling strategies and a description of the results we have obtained.

2 Parallel graph reduction

Execution of a functional program on GRIP is based on *graph reduction* [Pey87].

Specifically, our graph reduction model uses an abstract machine calleed the Spineless Tagless G-machine [Pey89]. The expression to be evaluated is represented by a graph of *closures*, each of which is held in heap. Each closure consists of a pointer to its *code*, together with zero or more *free-variable fields*. Some closures are in *head normal form*, in which case their code is usually just a return instruction. Other closures represent unevaluated expressions, in which case their code will perform the evaluation.

A closure is evaluated by jumping to the code it points to, leaving a pointer to the closure in a register so that the code can access the free variables; this is called *entering* the closure. When evaluation is complete, the closure is *updated* with (an indirection to) a closure representing its head normal form.

A *thread* is a sequential computation whose purpose is to reduce a particular sub-graph to normal form. At any moment there may be many threads available for execution in a parallel graph reduction machine; this collection of threads is called the *thread pool*. A processor in search of work fetches a new thread from the thread pool and executes it. A particular physical processor may execute a single thread at a time, or may split its time between a number of threads.

Initially there is only one thread, whose job is to evaluate the whole program. During its execution, this thread may encounter a closure whose value will be required in the future. In this case it has the option of placing of placing a pointer to the closure in the thread pool, where it is available for execution by other processors — we call this *sparking* a child thread.

If the parent thread requires the value of the sparked closure while the child thread is computing it, the parent becomes *blocked*. When the child thread completes the evaluation of the closure, the closure is updated with its normal form, and the parent thread is *resumed*. This blocking/resumption mechanism is the *only* form of inter-thread communication and synchronisation. Once an expression has been evaluated to normal form, then arbitrarily many threads can inspect it simultaneously without contention.

There are two main ways of organising the blocking/resumption process. In the *notification model*, the parent is blocked if the child thread has not completed when the parent requires its value, or if it has not started work (presumably because no processor was free to execute it). When the child thread completes it *notifies* the parent, which causes the parent to be resumed. The same applies to any other thread which is by then awaiting the value of the closure. The advantage of this model is that a parent can suspend execution with a notification count, awaiting notification from several children before it is resumed.

In the *evaluate-and-die model*, when the parent requires the value of a closure which it has sparked a child to evaluate, the parent *simply evaluates the closure just as if it had never created the child*. There are then three cases to consider:

- The child has completed its work, and the closure is now in normal form. In this case the parent's evaluation is rather fast, since it degenerates to a fetch of the value.
- The child is currently evaluating the closure. In this case the parent is blocked, and reawakened when the child completes evaluation. Upon reawakening the parent sees the closure in its normal form, and continues as in the previous case.
- The child has not started work yet. In this case there is no point in blocking the parent, and it can proceed to evaluate the closure. The child thread is still in the thread pool, but it has become an *orphan*, and can be discarded.

The evaluate-and-die model is the one we use in GRIP, for two main reasons. Firstly, blocking only occurs when the parent and child actually collide. In all other cases, the parent's execution continues unhindered without any context-switching overhead. In effect, the granularity has been dynamically increased by absorbing the child into the parent.

Secondly, since the parent will evaluate the sparked closure later, *it is legitimate for the run-time system to discard sparks altogether*. This opens up a very useful opportunity for load management, as we discuss in Section 5.1.

This concludes our overview of parallel graph reduction. Other useful references on the subject are [Eek88, LKI89, Gol88]. We now turn our attention to the GRIP architecture.

3 Architectural overview of the GRIP system

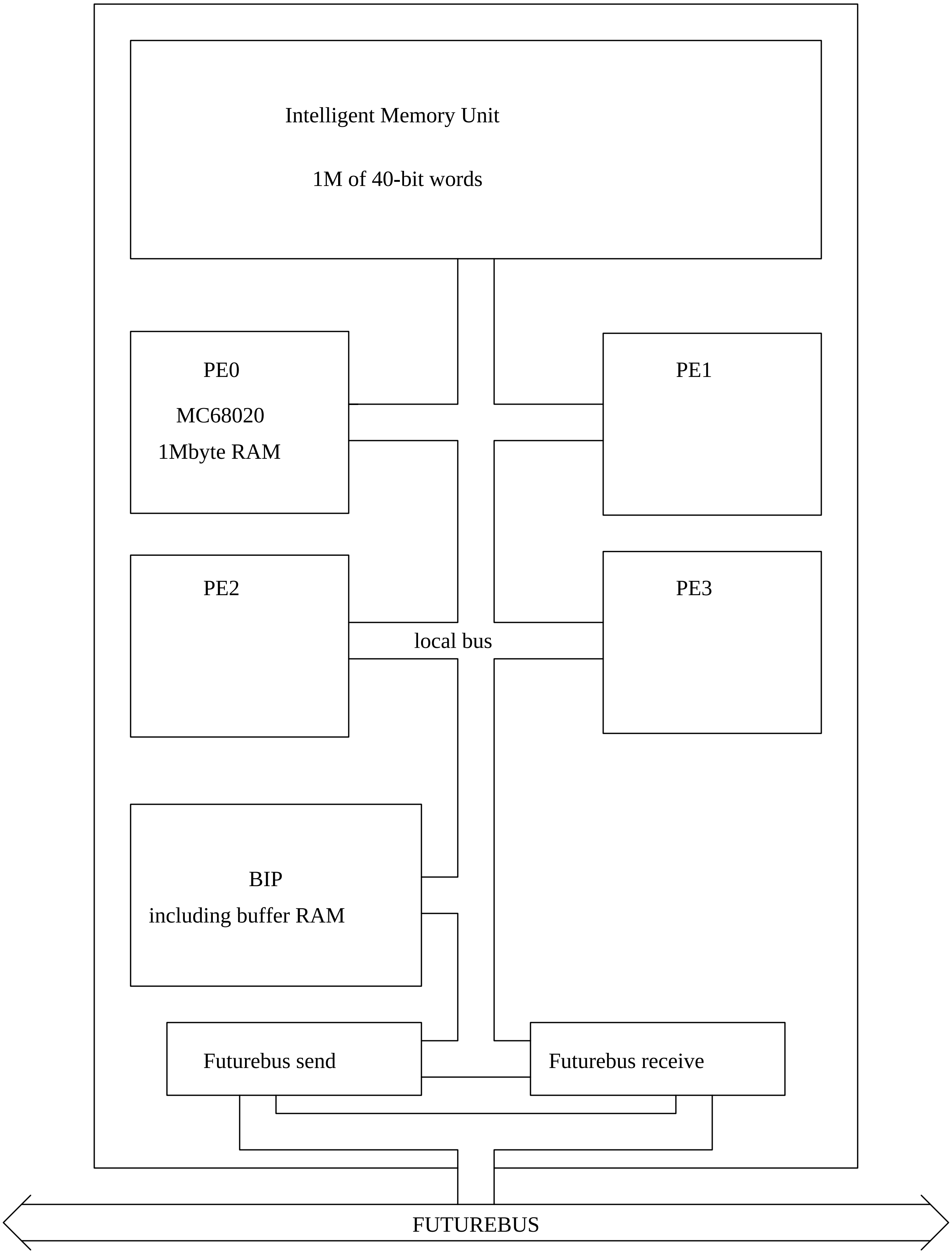

Figure 1: A GRIP board

GRIP consists of up to 20 printed circuit boards, each of which comprises up to four processors, one Intelligent Memory Unit (IMU) [JCSH87], and a communication subsystem (see Figure 1).

The boards are interconnected using a fast packet-switched bus [Pey86], and the whole machine is attached to a Unix host. A bus was chosen specifically to make the locality issue less pressing. GRIP still allows us to *study* locality, but does not require us to *solve* the locality problem before being able to produce a convincingly fast implementation. Communication between components on the same board is marginally faster than between components on different boards, but otherwise no difference in communication protocol is perceived by either the sender or recipient.

Each PE consists of a M68020 CPU, a floating-point coprocessor, and 1 Mbyte of private memory which is inaccessible to other processors.

The IMUs collectively constitute the global address space, and hold the graph. They each contain 1M words of 40 bits, together with a microprogrammable data engine. The microcode interprets incoming requests from the bus, services them and dispatches a reply to the sender. In this way, the IMUs can support a rich variety of memory operations, rather than the simple READ and WRITE operations supported by conventional memories.

The following range of operations is supported by our current microcode:

- Variable-sized heap objects may be allocated and initialised.
- Garbage collection is performed autonomously by the IMUs in parallel with graph reduction, using a variant of Baker's real-time collector [Bak78].
- Each IMU maintains a pool of executable threads. Idle processors poll the IMUs in search of these threads.
- Synchronised access to heap objects is supported. Heap objects are divided into two classes: evaluated and unevaluated. A lock bit is associated with each unevaluated object, which is set when the object is first fetched. Any subsequent attempt to fetch it is refused, and a descriptor for the fetching thread is automatically attached to the object.

  When the object is overwritten with its evaluated form (using another IMU operation), any thread descriptors attached to the object are automatically put in the thread pool by the IMU.

The IMUs are the most innovative feature of the GRIP architecture, offering a fast implementation of low-level memory operations with considerable flexibility. A separate project, called BRAVE, involves writing different microcode for the IMUs to support Prolog on the same hardware base [RK90].

The communications system and IMUs are quite fast. For example, it takes about $12\mu s$ for a one-word packet (eg a simple memory read) to be prepared by the PE, sent to an IMU, processed by the IMU (a few dozen microinstructions in this experiment), returned to the PE, and read by the PE. Of this $12\mu s$, most is spent by the PE code executed in the test, rather than in the communication latency or IMU processing time.

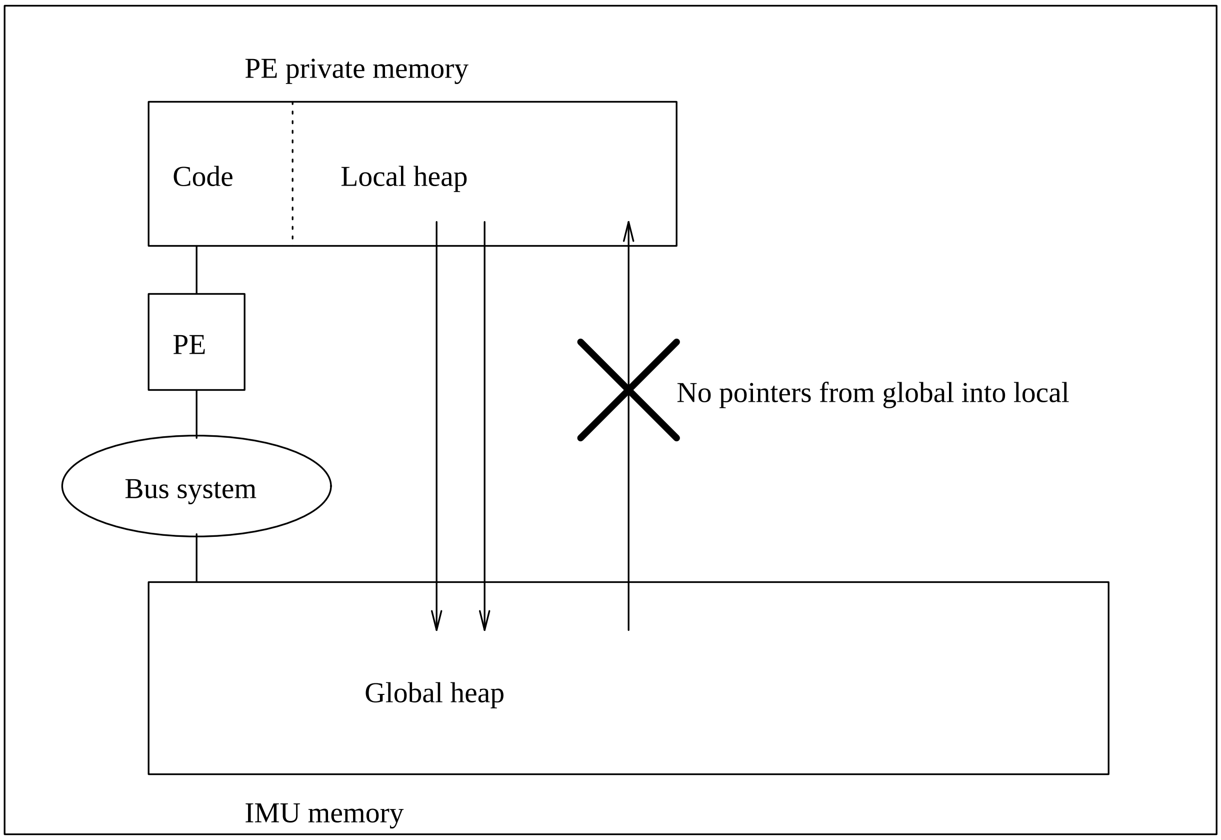

Figure 2: The two-level heap

4 Parallel graph reduction on GRIP

In this section we describe how we map the requirements of parallel graph reduction onto the resources provided by the GRIP hardware.

We start from the belief that parallel graph reduction will only be competitive if it can take advantage of all the compiler technology that has been developed for sequential graph-reduction implementations [Pey87]. Our intention is that, provided a thread does not refer to remote closures, it should be executed exactly as on a sequential system, including memory allocation and garbage collection.

We achieve this goal on GRIP in the following way (see Figure 2). The IMUs together hold the *global heap*, which is accessible to all PEs. A fixed part of the address space is supported by each IMU. In addition, each PE uses its private memory as a *local heap*, accessible only to that PE, in which it allocates new closures and caches copies of global closures. Each processor also holds a complete copy of the code for the program in its private memory.

We ensure that there are no pointers into the local heap from outside the processor, so it can be garbage-collected completely independently of the rest of the system. In contrast, when the IMUs run out of memory, the entire system is synchronised and global garbage collection is performed. Global garbage collections occur much less frequently than local garbage collections (by a factor of 100 or more).

Many closures are allocated, used, and garbage-collected without ever being copied into the global heap. In effect, local garbage collections recycle short-term "litter", while global garbage collection is required to recover data which has been shared. This effect is similar to that of a generational storage management system with just two levels, with two main differences. Firstly, the processor only has direct access to local-heap objects; global-heap objects are made accessible by caching a copy of them in the local heap. Secondly, closures are flushed into global memory for reasons other than increasing age, as we now discuss.

4.1 Moving closures between local and global heaps

When a processor needs a non-local closure, it fetches it from the appropriate IMU, and caches a copy in its local heap[2]. In the other direction, there are three reasons why a processor may decide to move a closure from local into global memory:

Making threads global. Each processor maintains a private thread pool of threads which it has sparked. Each such thread is just a pointer to a closure whose value will later be required. If the system load is low enough, it exports part of this pool to an IMU, remembering again to move into global memory the entire subgraph thereby made accessible. Again, the cost is a necessary one: threads which are being made public must be moved into a publicly visible place.

Nevertheless, this cost is only incurred when the system load is low. When the system fills up with threads, each processor simply runs locally.

Updating non-local closures. When a processor completes evaluation of a closure, it updates it with the evaluated head normal form. If the closure is a copy of a global one, the PE must also update the global copy. This is relatively expensive because, in order to maintain the invariant that there are no external pointers into a processor's private memory, *the entire subgraph accessible from the updated closure has to be moved into global memory* (at least, those parts which are not already there). This process is called *flushing* the subgraph.

Shortage of local heap space. When the local heap becomes full, extra space can be made in two ways. First, any local objects which are copies of global ones can be discarded; they will be re-fetched if they are needed again. Second, local objects which are not copies of global ones can be flushed to global memory and then discarded. Thus the system fails through memory exhausion only when the *global* heap becomes full.

Since flushing is expensive, it is clearly desirable to avoid performing sparks and updates whenever possible.

So far as sparks are concerned, a major topic of this paper is the avoidance of unnecessary sparks (Section 5.1).

Turning to updates, the Spineless Tagless G-machine (in common with some other abstract machines) allows the compiler to express on a case-by-case basis whether a closure requires to be updated. In particular, an update is unnecessary if the closure being updated is not shared.

[2]The fetch latency is such that it is not worth the overhead of context-switching while awaiting the result.

While this is a dynamic property of the graph, it is possible to derive an approximation to it at compile time. Two approaches look promising. Firstly, a static analysis can be made to detect sharing. This has the advantage of not requiring any changes to the program, but it is generally defeated by data structures. For example, even if a pair is not itself shared, its components may be when they are extracted somewhere else in the program.

A complementary new approach is based on *linear type systems* [Wad90, Abr90], whereby the type system of the program is used to express some of its sharing properties. Most proposals use a linear type system to avoid garbage collection (since an unshared object can be recovered immediately after its first use) [Wak90], but update avoidance looks like another promising application.

4.2 Coherence

Since each PE's local heap acts as a cache, containing *copies of* global closures, it is natural to ask how the thorny problem of multiprocessor cache coherence is dealt with. The answer is rather illuminating: the cache coherence problem does not arise in a parallel graph reduction system! There are two cases to consider:

- If a global closure is in head normal form, then it is immutable, and hence can freely be copied by as many PEs as wish to do so.
- If a global closure is not in head normal form, then the first to fetch it will *lock* it, and subsequent attempts to access it will be *blocked*, until the first PE updates it with its head normal form.

It follows immediately that cache incoherence cannot arise. This considerable architectural benefit derives directly from the declarative semantics of functional languages. The absence of side effects leads to a reduction of hardware cost, and an increase in performance, of the storage hierachy management system.

4.3 How it works

In this section we sketch how the two-level heap system is implemented on GRIP, focussing especially on the "fit" between it and the evaluation model used by the Spineless Tagless G-machine. No hardware support is required.

Each closure contains the following fields (Figure 3):

- Its code pointer.
- The address of the global closure of which it is a copy. If the closure has been locally allocated, and has no global counterpart, this field is zero.
- Zero or more fields for the free variables of the closure, each of which may or may not be a pointer.

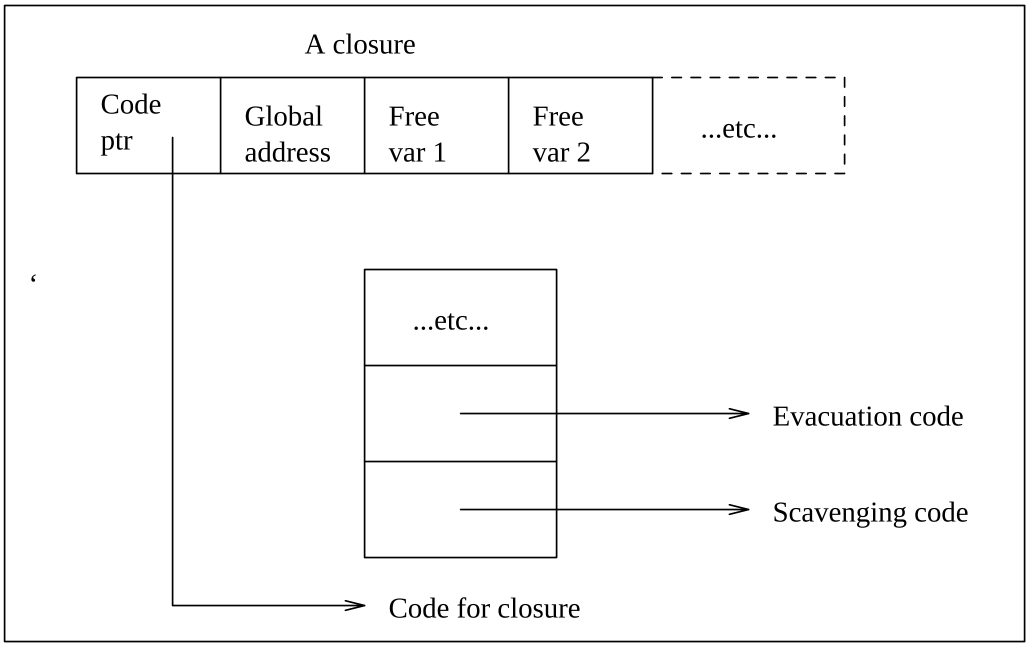

Figure 3: Accessing garbage-collection code from a closure

4.3.1 Fetching closures from the global heap

A potential source of overhead in the two-level heap is the need for the PE to distinguish between global and local pointers. It would indeed be a serious problem if, whenever the PE manipulated a pointer it had first to test a bit (say) to discover whether it was locally accessible or not! To explain how we avoid this cost, we digress for a moment to discuss the Spineless Tagless G-machine's evaluation model.

Suppose the value of closure is required. For example, the `append` function scrutinises its first argument to see if it is an empty list or a cons cell. If the closure is not yet evaluated it must be evaluated before being scrutinised.

This could be achieved by providing a flag on each closure to indicate whether it is in head normal form, and testing the flag just before the scrutinising code. Instead, we adopt the following convention: *a closure is always entered before being scrutinised.* (Recall that entering a closure amounts to loading a pointer to it into an environment register and jumping to its code.) If the closure is already evaluated, its code will simply be a return instruction, which will resume the scrutinising code. If not, its code will perform the evaluation, and return to the scrutinising code only when this is complete.

The advantage of this convention is that we can now add other special cases for free[3]. For example, when a closure is updated with its head normal form, what actually happens is that it is overwritten with an indirection to a closure representing the head normal form, which might be larger than the original closure[4]. No test need be made for an indirection, however, when scrutinising a closure, because when an indirection is entered, it simply enters the closure to

[3] As always, the term "for free" means "already paid for", but in this case the cost is fully justified!

[4] There are a number of special cases an optimisations which allow the original closure to be overwritten in place when it is certain that there is room to do so, but we won't disuss that further here.

which it points.

We are now ready to return to the question of distinguishing local from global pointers. The solution is now pretty obvious: use another form of indirection. Specifically, *a pointer to a global closure is represented by a pointer to a local closure, whose global address field points to the object, and whose code is called FETCHME.* When a FETCHME closure is entered, the FETCHME code copies the appropriate closure from global memory into the local heap, and updates the FETCHME closure with an (ordinary) indirection to the new closure. Any global pointers within the new closure are replaced with local pointers to new FETCHME closures.

In this way, closures are fetched lazily from global memory as and when required. Importantly, *in the common case of local graph manipulation*, it imposes no overhead at all.

This scheme suggests an interesting question: are graphs fetched as graphs, or are they unravelled as trees? Notice that the only issue here is saving (local) heap space, and communications bandwidth for the redundant fetches. There is no danger of duplicating computations in the original program, because of the coherence property mentioned above: only head-normal-form closures can be duplicated. It is easy enough to ensure that graphs remain as graphs, by having a hash table mapping global addresses onto local ones, but there is some cost associated with this. At present we have not implemented such a hash table, so graphs are unravelled. We plan to measure the impact of adding a hash table in due course.

4.3.2 Local garbage collection

Local garbage collection is performed using a conventional two-space copying collector. Its only unusual feature is the use it makes of precompiled code rather than interpretation.

A copying collector performs two operations on each object: it *evacuates* it from from-space into to-space, and it *scavenges* each object in to-space, which amounts to evacuating all the objects to which it points. We implement these operations by placing a table of code pointers immediately before the code pointed to by the closure (Figure 3). Two particular entries in this table point to code for evacuating and scavenging the closure. Since they are specific to the particular kind of closure, they know its exact structure, so they do not need to interpret layout information.

Furthermore, various special cases can be dealt with uniformly. For example, indirection closures have special-purpose evacuation code which "shorts them out" by instead evacuating the object to which they point. Similarly, forwarding pointers, which overwrite evacuated objects in from-space, have special-purpose code to return the address of the already-evacuated object. In short, the garbage collector performs almost no conditional tests, which makes it perform well.

4.3.3 Global garbage collection

For global garbage collection the IMUs also implement a two-space copying collector. They need to be told by the PEs the roots from which to collect. The PEs oblige by performing a local garbage collection, and sending the global-address field of every live closure (discovered during scavenging) to the appropriate IMU. The IMU responds with a new global address, with which the PE updates closure's global-address field.

The main interest in the global garbage collection algorithm is that it requires a distributed termination algorithm to detect completion. Even a single in-flight message from a PE to an IMU, telling it of a live closure, can give rise to an unbounded amount of further work. We use an algorithm derived from one designed by Dijkstra [DFG85].

4.4 Scheduling

In addition to managing the global heap, each IMU holds two pools of runnable threads:

- A thread is added to the *sparked thread pool* when a processor sends a spark message to the IMU, giving the address of a (global) closure whose value will subsequently be required. In fact, before doing so the IMU checks whether the closure referred to is in head normal form already, or is locked (indicating that it is under evaluation). In both these cases the thread is redundant, so it is discarded. Otherwise, it is added to the sparked thread pool.
- An thread is added to the *unblocked thread pool* when a closure is updated on which one or more threads are blocked. The blocked threads now become runnable, so are added to the unblocked thread pool. Unlike sparks, these threads must not be discarded.

When a PE runs out of local threads, it polls the IMUs until it finds some new ones to do. (It starts this process with the IMU on its own board, to enhance locality.)

The IMU returns an unblocked thread in preference to a sparked thread, if it has any. Unblocked threads represent partially-completed activities, which are best completed before starting new ones. In fact, the unblocked pool seldom seems to become large.

We make the usual assumption of *conservative parallelism*. That is, we assume that the value of sparked threads will eventually be required to produce the result of the program, and that threads are not wantonly sparked just on the off-chance that their value will prove useful. This assumption means that, to first order, it does not matter which thread from the pool we choose to execute, because all are required to get the final result. This is not the whole story, as we discuss in Section 5.2.

If non-essential (speculative) threads are present, the scheduler gets much more complicated. A priority system is required to make sure that progress is made on the non-speculative threads, and (harder still) some means of changing the priority of threads is required. (Thread priorities change when they become garbage (priority zero), or change from being speculative to being essential.)

Nevertheless, the conservative-parallelism constraint has a serious impact on expressiveness. It rules out, for example, programs which rely on parallel search which terminates with the first successful solution. For this reason, we plan eventually to relax this constraint in some carefully-controlled ways.

4.5 System management

As well as performing graph reduction, there are a number of system managment functions which GRIP must perform. As a result, each PE runs a small multitasking operating system called GLOS (GRIP Lightweight Operating System).

Each instance of GLOS supports one or more *tasks*. Most PEs run exactly one task, which performs graph reduction, but some run system management tasks which have a relatively low duty cycle, and therefore do not merit a processor to themselves. Specifically, there are the following system management tasks:

- The *Host Interface task* is the medium through which the host loads new tasks into the system and initiates them.
- The functional program is controlled by the *System Management task*, which coordinates the activities of the following three tasks, and the reduction task on each PE.
- The *Garbage Collection task* is responsible for the synchronisation of global garbage collection.
- The *Load Management task* controls global strategy for sparking and scheduling. These strategies are discussed in more details below.
- The *Statistics task* gathers statistics about the running of the functional program.

5 Sparking and scheduling strategies

In this paper we focus on two particular aspects of resource management, namely the decisions about *when to spark a thread*, and *which thread to schedule*. This section outlines the issues involved, while the following section describes the experiments we have performed.

5.1 When to spark a thread

The control of spark generation is critical: too few sparks will fail to exploit the parallelism offered by the machine, while too many will saturate the machine with many small threads. In the latter case, not only is the granularity of each thread small, but there is also a serious communications overhead, as the sparked threads have to be flushed into global memory so that they can be made visible to other processors. Finally, there is a danger that the machine's memory will be swamped with half-completed computations, thus preventing it completing any of them. *Throttling* the system by limiting the number of sparks generated may prove essential.

This is an aspect which has so far received little attention — existing work (e.g. [Sar87, SW89]) has focused mainly on load sharing rather than on thread creation. A notable exception is the MIT work on k-bounded loops [CA88], which restricts the number of concurrent iterations of a particular loop. This approach does not work in the more general setting of arbitrary recursive parallel programs, and we have tried a more general approach.

There are two times at which spark generation decisions can be made:

At compile time. All the opportunities for sparking are planted by the compiler. When the compiler finds a point where a closure can be sparked, it plants a call to the runtime system, passing a pointer to the closure.

A clever compiler with a strictness analyser might be able to insert sparks into an unannotated source program, but for the present we content ourselves with inserting annotations manually. So far our experience is that this is not hard to do, and not many are required.

At run time. We recall from Section 2 that the runtime system is free to discard sparks. Of course this risks losing parallelism, but it does provide an opportunity to regulate sparking based on the dynamic properties of the program running on a particular machine configuration, which is very hard to predict statically.

From now on we focus on runtime strategies for spark regulation. The goal is to *have as few sparks as possible, consistent with keeping all the processors busy.* On any machine, fewer sparks means larger threads and less communication. On GRIP, fewer sparks also means less flushing.

So far we have considered four main strategies.

Unregulated. Plenty of spark annotations in the source code, and no runtime regulation at all.

Cutoff. When the total pool of sparks exceeds a certain level, there is (usually) no point in creating further tasks; indeed, as pointed out above, there are significant costs associated with so doing. The Cutoff strategy is implemented by the Load Manager task, which regularly monitors the total size of the global spark pool, and issues a *spark rate* to each PE. This spark rate gives the number of sparks the PE is allowed to create in each clock tick. At its crudest, a spark rate of zero switches off all sparking, while one of infinity removes all spark restrictions.

Our present system uses two parameters to control the Load Manager. When the size of the spark pool increases to exceed the *spark cutoff level*, the Load Manager tells all PEs to stop sparking (ie discard all spark opportunities). When the pool size decreases again below the *spark switch-on level*, it tells them to start sparking again.

The intended effect to control the size of the spark pool to some specified level. Having two levels gives some hysteresis to avoid lots of flipping between the two states. So far, though, we have only made measurements with the two levels equal.

Delayed spawning. A common bad scenario is when a parent thread sparks a child, and then turns out to require the value of the child almost immediately. A typical example is the expression $e_1 + e_2$: suppose the parent sparks e_1, and then discovers that e_2 is either already in normal form, or very quick to evaluate. The parent now returns to e_1, and may well get blocked if another processor has started evaluating e_1 in parallel. Even if no other processor is evaluating it, a considerable cost has been paid to flush it into the global heap.

An idea which looks promising is to delay making the spark public for some fixed period. Then, if the parent returns to it quickly it will never be flushed, but if the parent does have lots of work to do, parallelism is not lost. Clearly the onset of parallel execution is delayed somewhat, but this may well be a price worth paying. We have not yet implemented this idea.

Hand tuning. For many programs it is possible to put more work into the annotations and thereby get a much better granularity. For example, in a divide-and-conquer algorithm, the programmer can cause sparks to happen only for the first n divisions (thus creating 2^n threads), hence matching the number of threads created to the number of processors available.

It would sad if it turned out that this was always necessary, but it does give a useful baseline against which to measure the more automatic techniques.

5.2 Which thread to schedule

When an IMU is asked by a PE for a sparked thread, it has a choice of which of the threads in its spark pool to return. The assumption of conservative parallelism means that the choice does not matter "much", because all threads contribute to the final result. But it is possible that a poor choice can degrade performance. For example, if all the theads are small except one large one, it would be a mistake to schedule the large one last!

A beautiful paper by Eager *et al* shows how to derive bounds on how good or bad the scheduling policy can be [EZL86]. They show that, under rather general assumptions, the worst case approaches the optimum as the average parallelism in the problem increases. When the average parallelism is equal to the number of processors, the worst case is only twice as bad as the optimum schedule.

Notwithstanding these results, it has been known for some while that the choice of scheduling policy has a significant impact on the size of the spark pool [BS81, RS87, Wat89]. Specifically, two strategies have been studied:

Return the most recently-sparked thread (LIFO). This strategy gives rise to a depth-first exploration of the process tree, which tends to limit the growth of parallelism.

Return the least recently-sparked thread (FIFO). In contrast, threads sparked earlier tend to correspond to larger computations, each of which itself contains more parallelism, so the FIFO strategy tends to give rise to a more rapid growth of parallelism.

This suggests that a FIFO strategy is appropriate when there are few tasks in the pool, switching to a LIFO strategy when the pool becomes fuller. In our system the Load Manager controls whether the IMUs follow a LIFO or FIFO scheduling strategy, based on its measurement of the overall load.

Somewhat surprisingly, in the absence of throttling, the choice of FIFO or LIFO strategy has at most a marginal impact on GRIP. We first realised this as a result of Deschner's simulation experiments [Des91], and then verified it on GRIP as we report in Section 6.3 below. However, these experiments do show that there is an interaction between LIFO/FIFO and other throttling strategies.

The effect is easily explained. Previous researchers have worked (in effect) entirely with the notification model for thread synchronisation (Section 5.1). That is, a parent thread sparks several children and then blocks awaiting their replies. This provides an opportunity to reschedule the processor running the parent. A LIFO strategy will tend to pick up one of the newly-sparked children right away, while a FIFO strategy will pick up an older thread.

By contrast, GRIP's evaluate-and-die model for thread synchronisation ensures that the parent thread will always execute one of the children itself, which effectively gives a LIFO strategy. The FIFO/LIFO decision is only made by processors which have become idle through being blocked, or completing a thread. Even so, the strategy does make some difference as we demonstrate below.

5.3 Measuring system load

Both the Cutoff sparking strategy and the LIFO/FIFO scheduling strategy depend on measurements of the overall system load made by the Load Manager. This measurement is rather imperfect, for two main reasons:

- It is always somewhat out of date, since there is no hardware support for computing it.
- It is based on counting the *number* of threads in the spark pools, rather than their *size*. Indeed, for many of them, by the time they are scheduled the closure to which they point may already have been reduced to head normal form by some other thread, in which case the spark has zero size[5].

One of the things we want to learn from our experiments is how much these imperfections matter. Is a crude load measurement good enough to deliver acceptable results?

6 Experimental Results

We have run a variety of experiments on GRIP. Previously we demonstrated absolute speedups for GRIP compared with an equivalent Sun workstation [HP90]. Here we report on results obtained using two of the simpler tests: naive Fibonnaci and a bitonic merge sort. The Fibonnaci program uses a regular divide-and-conquer algorithm producing a large number of extremely fine-grained computations (tens or hundreds of thousands of threads in the typical case). It is thus an effective "stress test" for our load-balancing strategies. Bitonic merge sort uses an irregular divide-and-conquer algorithm to sort an array of values represented by a binary tree. It also generates many fine-grained threads, but these threads are rather larger than for Fibonacci.

[We have run GRIP on various more interesting programs than these two, and the final paper will report on this data.]

The timings reported here were obtained from GRIP's internal clock, which has a resolution of 1ms. Timings were taken from the start of reduction to the receipt of the first "kill" signal by the system manager. Unless otherwise stated, all tests were run on an 18-PE GRIP with 7 IMUs.

[5] This case can be detected by the IMU, which then simply discards the spark rather than returning it to a PE.

6.1 The spark cutoff policy

Our first experiment tests the effectiveness of the Cutoff sparking policy (see Section 5.1), by varying the spark cutoff setting while keeping the spark rate constant. Figure 4 shows the timing results obtained from the Bitonic merge sort example for a spark cutoff level varying between 1 and 100000, at a constant spark rate of 15 sparks per PE per millisecond clock tick. Clearly, the spark cutoff setting has little effect on the overall runtime for this example. Equally clearly, there is a wide variation in the timings which results from a single cutoff value. This is disappointing since we anticipated that the cutoff could be set heuristically to provide good, consistent timings for a given application.

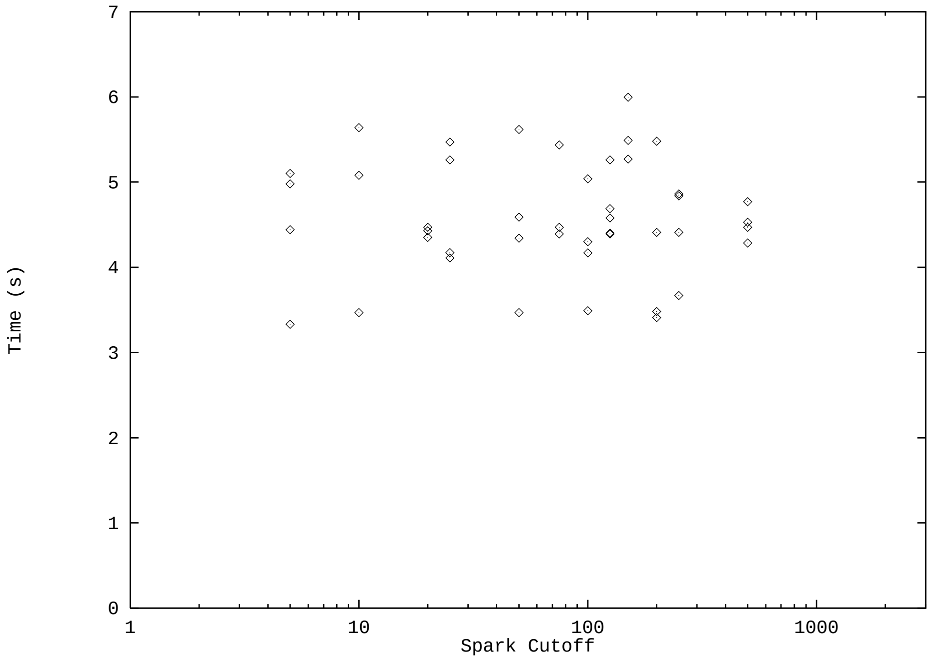

Figure 4: Bitonic Merge Sort, Spark Rate = 15

Figure 5 gives more detailed picture of one particular run, with a spark cutoff of 100 sparks. The upper graph shows the processor activity in three categories: reducing, idle, and communication (reading and flushing). The lower graph shows how sparks are generated and consumed. The solid line indicates the number of sparks in each time period; the dashed line shows the number of sparks taken from the thread pool in the time period; and the dotted line shows the number of blocked threads which are resumed.

There are oscillations in the reduction rate, and significant variations in the sparking rate. The large number of resumed threads indicates that many threads are blocked awaiting some result to be computed, that is that the data inter-dependency is high.

We conclude that, in this case, the Load Manager is not responding sufficiently rapidly to changes in the system load — it only samples the state of the system every 1ms. This is an important result. We have a number of as-yet unimplemented ideas for improving the load management policy to avoid this erratic behaviour. Firstly, each PE can keep a local pool of hundreds of sparks (which are cheap, because have not been flushed). Secondly, the Load Manager can give

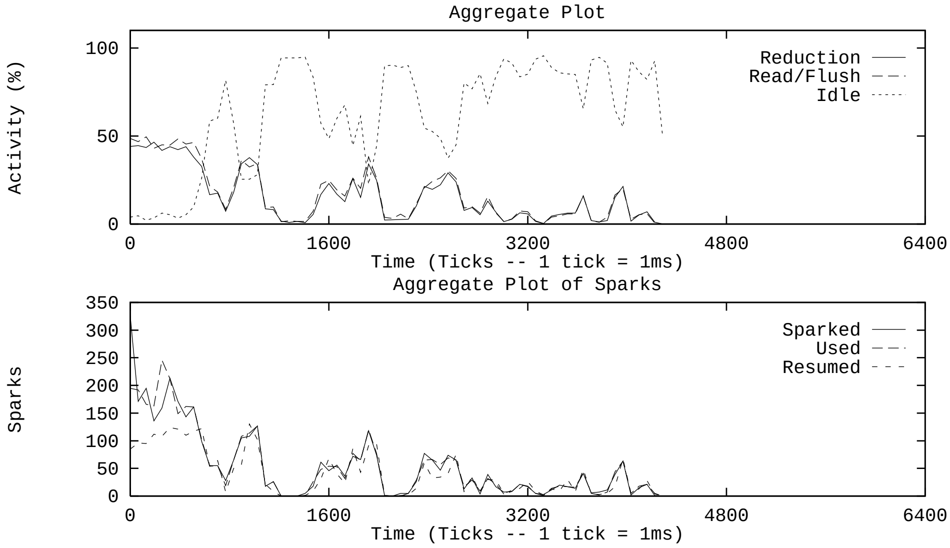

Figure 5: Bitonic Merge Sort, Aggregate Activity and Spark Profiles

each IMU a target thread pool size; each IMU can then negotiate with its local processors to flush some of their local thread pool when necessary. What is clear is that control of the thread pool size must be devolved more than at present.

6.2 Varying the spark rate

For our next experiment, we vary the spark rate while keeping the spark cutoff constant. Figure 6 shows the timings for a parallel Fibonacci program with a cutoff of 50. Although the curve is not smooth, it is clear that low (< 10) and high (> 1000) spark rates are both detrimental to the overall runtime performance, with the best timings occurring at spark rates between these values. This is a result we expect: at low spark rates, starvation is likely; conversely, at high spark rates, congestion will occur.

6.3 LIFO/FIFO scheduling strategy

Figure 7 compares a fixed LIFO thread pool strategy, against a fixed FIFO thread pool strategy with no spark cutoff. This demonstrates the interesting effect discussed in Section 5.2: GRIP is insensitive to the thread pool strategy in use. In fact, pure LIFO scheduling is marginally less effective than pure FIFO scheduling for all spark rates, with the spark cutoff set at infinity. This result is an artefact of GRIP's evaluate-and-die synchronisation policy, discussed earlier.

Interestingly, the LIFO/FIFO decision and seems to be much more important in conjunction with the spark cutoff policy. Figure 8 shows the same program with a spark cutoff level of 1000 sparks. The LIFO strategy shows far less variation than the FIFO strategy, and is also consistently faster (and considerably faster than either policy without a cutoff at all). This is consistent with results from the literature on dataflow machines.

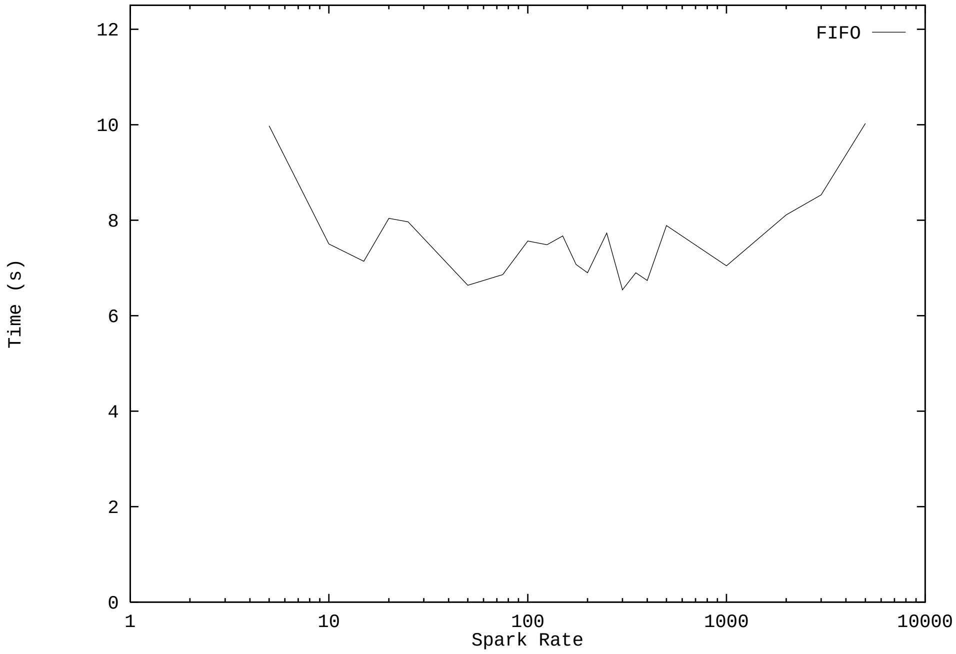

Figure 6: Fibonacci, Spark Cutoff = 50

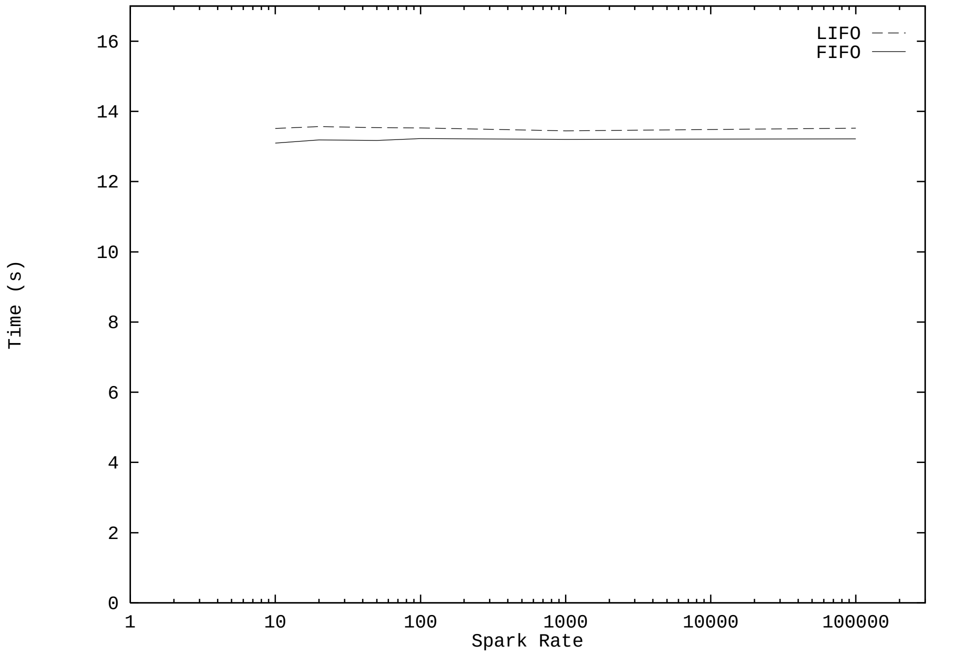

Figure 7: Fibonacci, LIFO v. FIFO, Cutoff = ∞

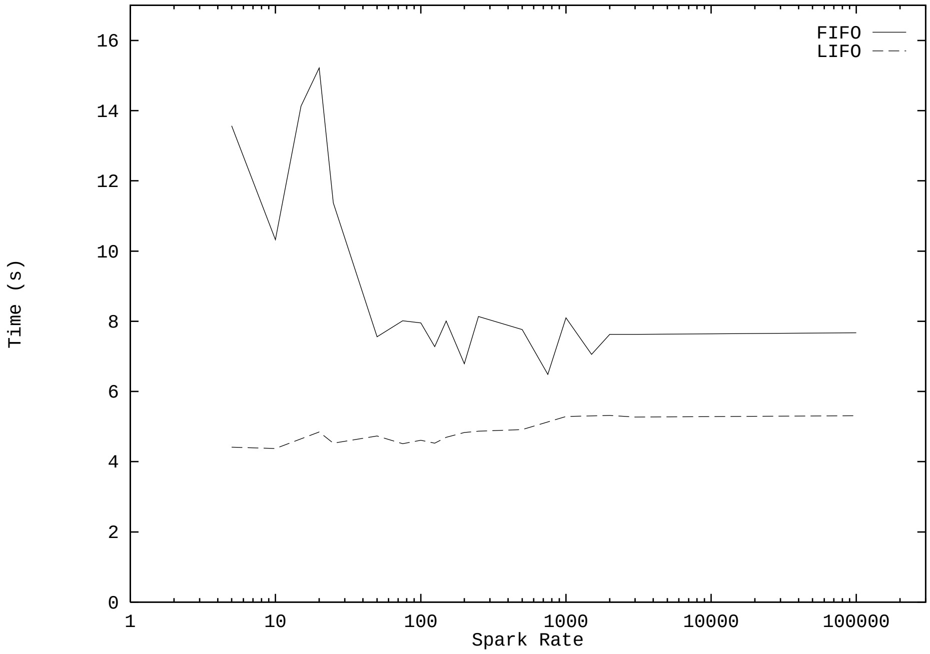

Figure 8: Fibonacci, LIFO v. FIFO, Cutoff = 1000

6.4 Varying the number of processors

In Figure 9, we demonstrate the speedup obtained as we add PEs, for the Fibonacci program. As mentioned earlier, because the granularity is very fine this is a real stress test for an automatically-scheduled system.

We plot not only the total times, but also the absolute times spent in reduction and communication, together with the idle time. This gives a measure of the effectiveness of our throttling strategy. We observe that the PEs spend approximately 65% of their time reducing, 25% communicating with the IMUs, and <10% idle. Garbage collection (not shown on this graph) accounts for 2-3% of the PEs' time. Overall, the 18-PE system is 9 times faster than a sequential version of the program.

Figure 10 gives a more detailed look at a single run, showing the processor activity and spark profiles. Overall, about 64% of the PEs' time is spent reducing, 25% communicating (reading/flushing) and 9% is spent idle. The remainder (2%) is garbage collection time. It is possible to observe a correlation between the aggregate spark rate and the communication time, as we would expect. The peaks of spark usage probably correspond to situations where many small threads are executed in a short space of time. The drops in the number of sparks generated, which occur simultaneously with these peaks, supports this hypothesis (each thread used is generating very few new sparks).

Finally, Figure 11 shows yet more detail from the same run, by giving the activity graphs for four of the individual PEs. Loss of reduction time is accounted for almost entirely by an increase in communication. From these results and those cited earlier it is clear that significant performance improvements can only be achieved by concentrating on reducing communication overhead, either by more restrictive throttling, as with the local scheme we tried previously

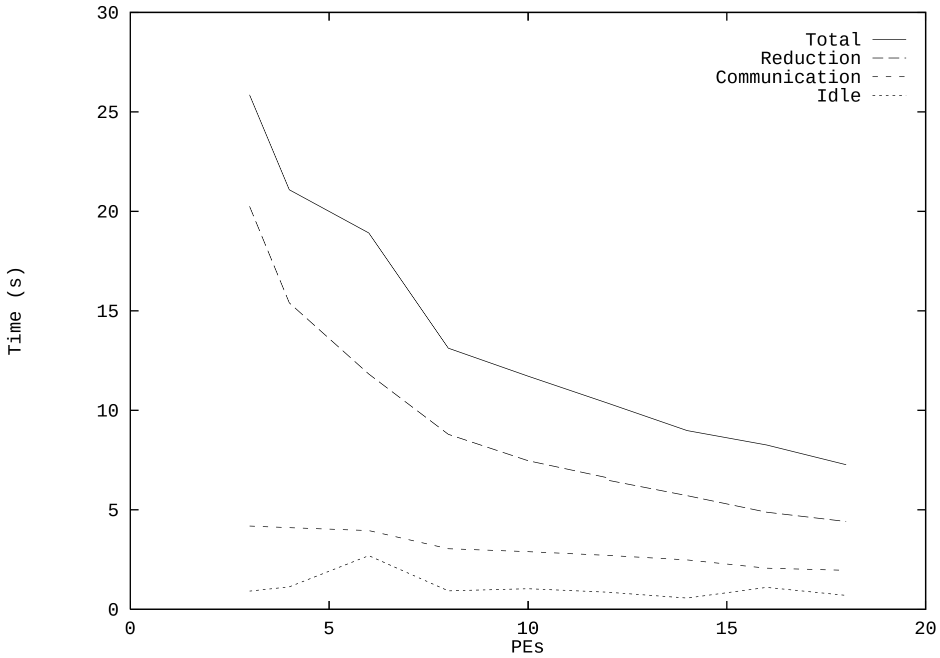

Figure 9: Fibonacci, Spark Rate = 25, Spark Cutoff = 50

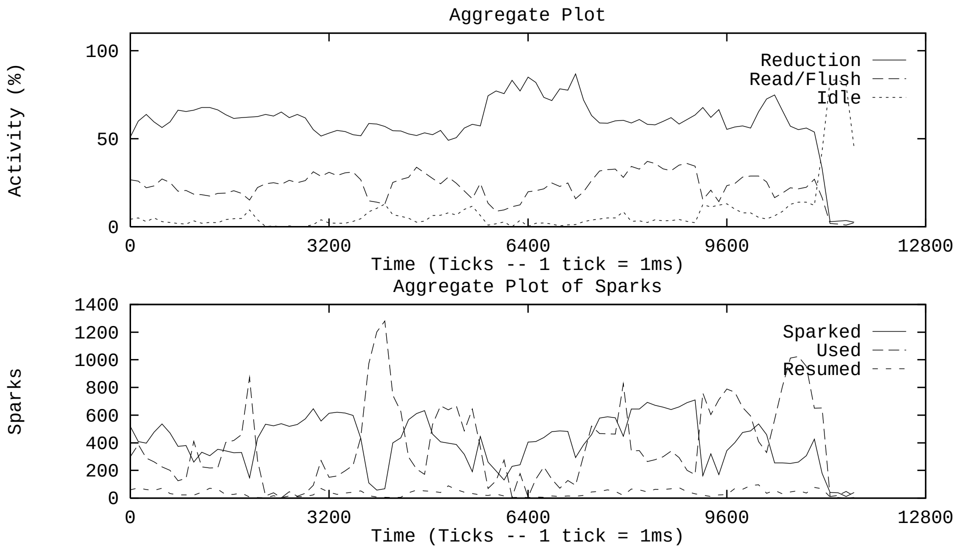

Figure 10: Fibonacci, Spark Cutoff = 50, Aggregate Activity and Spark Profiles

[HP90], or by using a scheme where sparked closures are flushed to global memory only on demand as suggested in Section 6.1.

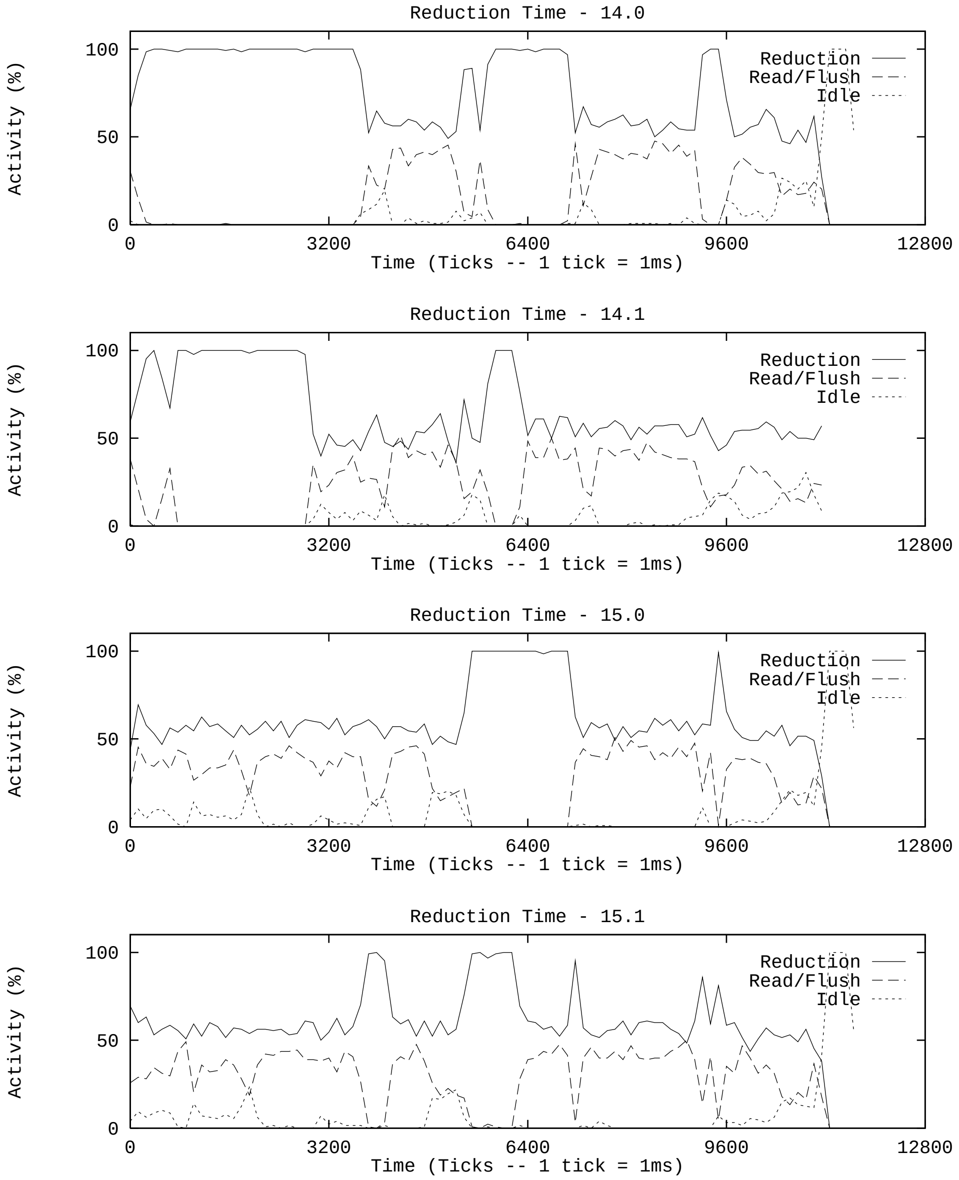

Figure 11: Fibonacci, Spark Cutoff = 50, Reduction Profiles

These results are encouraging. We are lose only a factor of two (compared with perfect linear speedup) by handing over almost complete control to the system, and accepting very small threads. Our work is still at an early stage, and we are confident that we can improve this figure by tuning the scheduling heuristics, still without requiring programmer intervention. A programmer who wanted to improve matters still further could take more care not to spark small tasks. It remains to be seen whether we can reprodue these results for other programs.

6.5 Varying the number of IMUs

Another interesting question is how sensitive the system is to the number of IMUs. Figure 12 shows that varying the number of IMUs in the system has little effect on overall runtime performance. Response is slightly better if the number of IMUs is increased (probably reflecting decreased packet queuing), but this difference does not appear to be significant. This is a

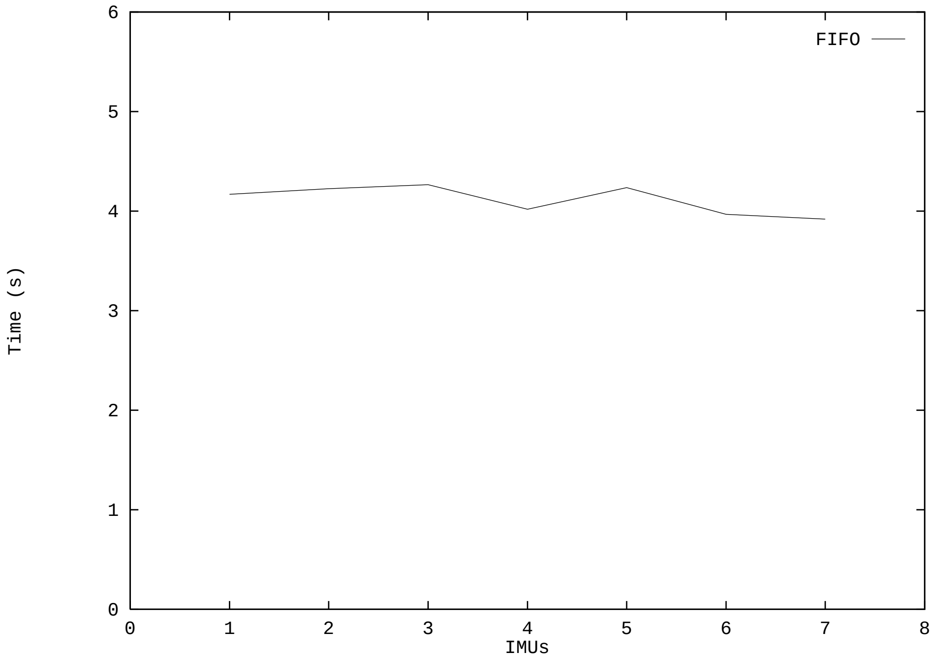

Figure 12: Bitonic Merge Sort, Spark Rate = 15, Spark Cutoff = 25

rather satisfactory result. Even one IMU seems quite capable of servicing all 18 PEs without significantly holding up the system. The IMUs seem unlikely to become a bottleneck.

7 Conclusions

We have presented the results of running several experiments on the GRIP parallel reducer, a purpose-built machine designed to execute functional programs efficiently. Our results lead us to draw three main conclusions:

- It is possible to execute at least some fine-grained programs on GRIP, without incurring more than a small constant factor overhead in performance. (Indeed, such a program may have better performance than a badly-partitioned coarse-grained version of the same algorithm.)
- For GRIP, there is no significant difference between LIFO and FIFO scheduling if no sparks are discarded. However, LIFO scheduling often gives better performance when a spark-discarding load manager is used.

- To be effective, the load management system must respond more rapidly to important changes in the system (perhaps even anticipating these changes and taking appropriate preemptive action).

8 Further Work

Several strands remain to be explored. Most obviously, the results presented here consider only trivial programs. We believe that experiments with larger programs, such as the partial evaluator and the theorem prover we have compiled for GRIP, will support these results, but further rigorous experimentation is needed.

Our results have highlighted the primary deficiency of the current load manager: its instability under rapidly changing conditions. One obvious solution is to devolve certain load management tasks from the load manager to the IMUs. If the IMUs are each provided with a target thread pool size, then they may actively reject unwanted sparks, or stimulate the PEs to produce more sparks if the thread pool is empty. We are reluctant to simply increase the frequency with which the load manager runs, since this could lead to bus saturation or other performance bugs if not carefully controlled.

Our results also show in a graphic way the overhead associated with flushing closures to memory when they are sparked. By maintaining a pool of potential sparks locally in each PE, it may be possible to avoid flushing closures unless some other PE is capable of evaluating the closure. This local pool might even be fetched preemptively by an IMU if the size of its thread pool fell below some threshold.

Although we have implemented hysteresis in the spark cutoff policy in our system (Section 5.1, we have not yet conducted significant experiments using this mechanism (principally because the slackness in the load management loop would make the effect of hysteresis unpredictable). We intend to experiment with this mechanism once we have improved the feedback loop in the load manager.

References

[Abr90] Abramsky S, "Computational interpretations of linear logic", *DOC 90/20, Dept of Computing, Imperial College, London*, 1990.

[AE87] Arvind and Ekanadham K, "Future scientific programming on parallel machines", *Proc International Conference on Supercomputing, Athens*, June 1987.

[Bak78] Baker HG, "List processing in real time on a serial computer", *Comm ACM* 21(4), April 1978, pp. 280-294.

[Bur89] Burton FW, "Encapsulating nondeterminacy in an abstract data type with determinate semantics", *Journal of Functional Programming* 1(1), Jan 1991, pp 3-20.

[BS81] Burton FW and Sleep MR, "Executing functional programs on a virtual tree of processors" *Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire*, Oct 1981, pp 187-194.

[CA88] Culler D and Arvind, "Resource requirements of dataflow programs", *Proc 15th Annual Symposium on Computer Architecture*, ACM, 1988.

[Des91] Deschner J, "Simulation of parallel graph reduction", *MSc thesis*, Dept of Computing Science, University of Glasgow, 1991.

[DFG85] Dijkstra EW, Feijen WHJ and van Gasteren AJM, "Derivation of a termination detection algorithm of distributed computations", *in Control Flow and Data Flow - Concepts of Distributed Programming*, ed. Broy, LNCS Springer Verlag, 1985.

[EZL86] Eager DL, Zahorjan J and Lazowska ED, "Speedup versus efficiency in parallel systems", *Dept of Computational Science, University of Sasketchewan*, 1986.

[Eek88] M van Eekelen, "Parallel graph rewriting", *PhD thesis*, University of Nijmegen, 1988.

[FW87] Fairbairn J, and Wray S, "TIM - a simple lazy abstract machine to execute supercombinators", *Proc FPCA 87*, Portland, Oregon, ed Kahn G, LNCS Springer-Verlag, 1987.

[Gol88] BF Goldberg, "Multiprocessor execution of functional programs", *PhD thesis*, YALEU/DCS/RR-618, Dept of Computer Science, Yale University, April 1988.

[HP90] Hammond K, and Peyton Jones SL, "Some Early Experiments on the GRIP Parallel Reducer", *Proc 2nd Intl Workshop on Parallel Implementation of Functional Languages*, ed Plasmeijer MJ, University of Nijmegen, 1990.

[Hud86] Hudak P, "Para-functional programming", *IEEE Computer 19(8)*, Aug 1986, pp 60-71

[HW90] Hudak P, Wadler PL *et al*, "Report on the functional programming language Haskell", *Dept of Computer Science, Glasgow University* April 1990.

[HOD89] Hughes RJM and O'Donnell JT, "Expressing and reasoning about non-deterministic functional programs", *Functional Programming, Glasgow 1989*, ed Davis and Hughes, Springer Verlag Workshops in Computing, 1989.

[JCSH87] Peyton Jones SL, Clack, C, Salkild, J and Hardie, M "GRIP – a high-performance architecture for parallel graph reduction", *Proc FPCA 87*, Portland, Oregon, ed Kahn G, Springer-Verlag LNCS, 1987.

[Joh84] Johnsson T, "Efficient compilation of lazy evaluation", *Proc SIGPLAN Symposium on Compiler Construction, Montreal*, June 1984.

[LKI89] R Loogen, H Kuchen, K Indermark, and W Damm, "Distributed implementation of programmed graph reduction", *Parallel Architectures and Languages Europe*, LNCS 365, Springer Verlag, June 1989.

[Pey86] Peyton Jones SL, "Using Futurebus in a fifth generation computer", *Microprocessors and Microsystems*, March 1986.

[Pey87] Peyton Jones SL, *The implementation of functional programming languages*, Prentice Hall, 1987.

[Pey89] Peyton Jones SL, and Salkild J, "The Spineless Tagless G-machine", *Proc FPCA 89*, London, ed MacQueen, Addison Wesley, 1989.

[RK90] Reynolds TJ and Kefalas P, "OR-parallel Prolog and search problems in artificial intelligence applications", *7th International Conference on Logic Programming*, pp 340-354, Jerusalem, June 1990.

[Roe91] Roe P, "Parallel programming using functional languages", PhD thesis, *Department of Computing Science, University of Glasgow*, Technical report CSC 91/R3, 1991.

[RS87] Ruggerio CA, Sargeant J, "Control of parallelism in the Manchester dataflow machine" *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland* ed Kahn G, Springer Verlag LNCS 274, Sept 1987, pp. 1-15.

[Sar87] Sargeant J, "Load Balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines", *Internal Report UMCS-86-11-5*, Department of Computer Science, University of Manchester, January 1987.

[SW89] Siedl H, and Wilhelm R, "Probabilistic Load Balancing for Parallel Graph Reduction", *Proc IEEE TENCON 89*, Bombay, India, Novermber 1989, pp. 879-884.

[Wad90] Wadler P, "What's the use of linear logic", *Dept of Computing Science, University of Glasgow*, 1990.

[Wak90] Wakeling D, "Linearity and laziness", *PhD thesis, University of York*, 1990.

[Wat89] Watson I, "Simulation of a physical EDS machine architecture", *Internal report, Dept of Computer Science, University of Manchester*, 1989.