A Parallel Functional Database on GRIP

Gert Akerholt Kevin Hammond Simon Peyton Jones Phil Trinder*

June 4, 1991

Abstract

GRIP is a shared-memory multiprocessor designed for efficient parallel evaluation of functional languages, using compiled graph reduction. In this paper, we consider the feasibility of implementing a database manager on GRIP, and present results obtained from a pilot implementation. A database implemented in a pure functional language must be modified *non-destructively*, i.e. the original database must be preserved and a new copy constructed. The naive implementation provides evidence for the feasibility of a pure functional database in the form of modest real-time speed-ups, and acceptable real-time performance. The functional database is also used to investigate the GRIP architecture, compared with an idealised machine. The particular features investigated are the thread-creation costs and caching of GRIP's distributed memory.

1 Introduction

Multiprocessor machines potentially provide considerable computing power cheaply. Because they typically have large memories, multiprocessors are a particularly attractive choice for implementing data- or transaction-processing applications. Pure functional languages are easily made concurrent and hence are particularly suitable for programming multiprocessors. Indeed, there are now several declarative multiprocessors, i.e. machines specifically designed to evaluate functional languages in parallel [Alv85, BS81, CGR89, Pey86].

There are many research issues which remain to be resolved for multiprocessor machines, for example locality, granularity, throttling and load balancing. Data processing poses an additional challenge for declarative systems because data structures must be modified *non-destructively*, that is, the original structure must be preserved and a new copy constructed. Because of the high cost of non-destructive update the designers of the Flagship declarative multiprocessor provided side-effecting transactions as operating-system primitives. The Flagship machine has achieved some impressive transaction-processing results, being approximately twice as fast as both a large VAX (an 8830) and a large IBM (a 4381-P22) [Rob89]. However, the side-effecting transactions made the semantics of programs complex, reasoning about programs difficult and evaluation-order significant.

An alternative purely-functional approach based on creating versions of trees was proposed in [AFHLT87]. Essentially only a path from the root to the site of the update needs to be duplicated

*This work is supported by the SERC GRASP and Bulk Data Types Projects. Authors' address: Computing Science Dept, Glasgow University, Glasgow, Scotland. Email: {akerholg,kh,simonpj,trinder}@uk.ac.glasgow.cs

— the unchanged parts can be shared by the original tree and the new tree. Encouraging results were obtained for the simulated parallel evaluation of a prototype database using this approach [Tri89, Tri90].

Preliminary results obtained from a prototype database implemented on the GRIP declarative multiprocessor are reported here. GRIP is intended to execute lazy functional languages efficiently in parallel. Two issues are investigated by the GRIP implementation. Since the implementation is the first pure functional database on a true multiprocessor, it provides insights into the feasibility of implementing a transaction manager in a pure, i.e. non-destructive manner. The second issue is whether the GRIP machine provides good real-time performance, in particular whether parallel evaluation of the database can produce real-time speed-ups.

From an architectural perspective, the database application is interesting in two respects:

- it bridges the gap between simple, well-understood parallel benchmarks such as *nfib*, and complex real functional programs such as the Veritas theorem prover. The parallel behaviour of the database is sufficiently understood for us to be able to draw useful conclusions about the performance of GRIP, yet it is complex enough to provide a satisfactorily realistic example program.
- it tests our policy of dynamic data distribution, and provides a benchmark we can exploit for measuring realistic local memory cache performance.

The results show that both lookup and update transactions can be handled efficiently in parallel, with real absolute speed-ups. The latter is particularly important – it is pointless to merely achieve good relative speed-ups. The costs of communication is also highlighted by this example, often approximating the reduction costs. This prompts us to suggest some architectural improvements.

The remainder of the paper is structured as follows. Section 2 provides an overview of GRIP's architecture and that of the simulated machine. Section 3 describes the architecture of our database application, introducing the data manager and the mechanisms for generating parallelism. Section 4 presents comparative performance results for both GRIP and a simulated ideal machine. Section 5 concludes.

2 Machine Architectures

2.1 GRIP Architecture

2.1.1 Overview

GRIP consists of up to 20 printed circuit boards, each of which consists of up to four processing elements (PEs) and one Intelligent Memory Unit (IMU). 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 faster than between components on different boards, but otherwise no difference in communication protocol is perceived by either the sender or recipient.

Each unit which communicates with the bus therefore needs a fairly sophisticated bus interface processor (BIP) to store packets and forward them to their destination when the bus becomes available. A single BIP is shared between the IMU and the PEs.

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

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 bus. In this way, the IMUs can support a 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 IMU microcode:

- Variable-sized heap nodes 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 PEs poll the IMUs in search of these threads.
- Synchronised access to graph nodes is supported. A lock bit is associated with each unevaluated node, which is set when the node is first fetched. Any subsequent attempt to fetch it is refused, and a descriptor for the fetching thread is automatically attached to the node.

  When the node is overwritten with its evaluated form (using another IMU operation), any thread descriptors attached to the node 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 great flexibility.

2.1.2 Graph reduction on GRIP

Our graph reduction model is based on the Spineless Tagless G-machine [PS89]. The expression to be evaluated is represented by a graph of *closures*, each of which is held in heap nodes. Each closure consists of a pointer to its *code*, together with zero or more *free-variable fields*. Some closures are in *normal form*, in which case their code is usually just a return instruction, while others 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 normal form.

A *thread* is a sequential computation whose purpose is to reduce a particular sub-graph to weak head 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 PE in search of work fetches a new thread from the thread pool and executes it. A particular physical PE may execute a single thread at a time, or may divide 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 PEs — 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*.

We use the evaluate-and-die model of blocking/resumption, as described in [PS89, HP90]. The advantage of this model is that blocking only occurs when the parent and child actually collide. In all other cases, the parent's execution continues unhindered.

Notice that in either case the 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. This synchronisation provides the inter-transaction "locking" required by the functional database, as described in [Tri89]. A transaction demanding the result of a previous transaction is blocked until the previous transaction has constructed the value it requires.

2.1.3 Mapping parallel graph reduction onto GRIP

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 [Pey87a]. Our intention is that, provided a thread does not refer to remote graph nodes, it should be executed exactly as on a sequential system, including memory allocation and garbage collection.

We implement this idea on GRIP in the following way. The IMUs together hold the *global heap*, a fixed part of the address space being supported by each IMU. Each PE uses its private memory as a *local heap*, in which it allocates new graph nodes and caches copies of global heap nodes. We ensure that there are no pointers into this local heap from outside the PE, so it can be garbage-collected completely independently of the rest of the system. When the IMUs run out of memory, the entire system is synchronised and global garbage collection is performed. Global garbage collection happens much less frequently than local garbage collections. In effect, local garbage collections recycle short-term "litter" without it ever becoming part of the global heap.

2.2 Metrics

We currently monitor two kinds of performance data: processor activity and spark usage. These are plotted as profiles against elapsed time. Although we also record total heap usage, we are not yet able to plot heap occupancy profiles.

Since garbage collection overheads are low, and the total available memory is high (128Mbyte for a fully-configured GRIP, or 42Mbyte in the system monitored here), space usage is not reported here. Obviously, the rate of heap consumption will affect the size of database which can reasonably be supported, but we believe that this is a secondary issue at present.

2.3 Processor activity

We break down the distribution of processor time into the following categories:

- Reduction: this corresponds almost exactly to what a sequential Spineless Tagless machine would do.
- Communication: this is almost entirely due to *flushing* out parts of the graph into global memory and *reading* other parts in.
- Idle: time spent looking for work.
- Garbage collection.
- Other overheads: time spent running other GLOS tasks, or performing GRIP to host IO. This is always negligible.

The figures for communication time include the time taken to construct packets, in addition to the elapsed communication time. Measurements show that packet construction time generally exceeds the raw communication time, at least for a lightly loaded system.

As the program runs, each PE accumulates a profile of this breakdown against time, which is collected by the host when the program completes. Since the activities are interleaved quite finely, it is not adequate to refer to a real-time clock at the transition between activities. Instead, whenever there is a clock interrupt, we inspect a global variable which tells what the PE is currently doing, and increment an appropriate count. We maintain these counts in an array of 100 "buckets", each corresponding to a fixed time interval. When we run out of buckets, we combine the counts in adjacent buckets and double the time interval corresponding to a bucket. In this way we always have data with a time resolution of better than 2%, without having to decide any parameters in advance. Its precision within each bucket naturally depends on how many clock ticks correspond to a bucket. Each clock tick represents 1ms of elapsed time. The breakdown is plotted both on a processor-by-processor basis, and aggregated into a single plot.

2.4 Spark profiles

We accumulate a time profile of how sparks are generated and used, in three categories:

Sparks generated. This count is incremented whenever a processor sparks a new thread.

Sparks used. Incremented whenever a processor gets a new thread to execute. There will often be fewer of these than sparks generated, because of the ones discarded because they turn out to be in normal form or under evaluation by the time they are selected for execution.

Threads resumed. This is incremented whenever a processor resumes execution of an old thread, which had been blocked but which has now been freed for execution.

These profiles are accumulated in the same bucket array as the processor-activity profiles, and dynamically rescaled in the same way. We also plot IMU sparking profiles, and record the overall totals for sparking, the thread pool size etc.

2.5 The Simulated Machine Architecture

2.5.1 Overview

The hypothetical machine is implemented by a interpreter simulates parallel graph reduction. The hypothetical machine is intended to be an ideal declarative multiprocessor and it is interesting to compare GRIP's performance results with the results from such a machine. Fuller details of the architecture and instrumentation can be found in [Tri89]. We simulate a MIMD shared-memory architecture.

The machine performs super-combinator graph reduction [Pey87a]. The evaluation strategy used in the machine is lazy except where eagerness is introduced by the primitives described in the following Section. In a machine cycle a processing element may either

- Perform a super-combinator reduction, or
- Perform a delta-reduction, i.e. evaluate a primitive such as 'plus', or
- Perform a house-keeping activity such as sparking a new thread.

The work to be performed by the program is broken into threads. Each thread reduces a subgraph of the program graph. Initially only one thread exists. New threads are sparked by the eager primitives described later. Thread synchronisation occurs as follows. A thread marks the nodes it is processing as busy. A thread encountering a busy node is marked as *blocked*. As soon as the required node is no longer busy the blocked thread resumes. A thread that is not blocked is termed *active*. The scheduling strategy used in the hypothetical machine is both simple and fair: every active thread is assigned to a processing agent and in a machine cycle the next redex in each active thread is reduced.

The machine is simplistic in not supporting throttling or granularity control, in having a uniform machine cycle and in reducing newly sparked threads immediately.

The hypothetical machine is instrumented to record the following statistics during the evaluation of a program: the number of super-combinator and delta- reductions, the number of graph nodes allocated, the number of machine cycles and the average number of active processes. In addition, every 20 machine cycles, the hypothetical records the average number of active threads. This information is used to plot graphs of the average number of active threads against time measured in machine cycles.

3 Database Architecture

3.1 Introduction

A *class* is a homogeneous set of data; it may represent a semantic object such as a relation or an entity set. For example a class might represent a collection of bank accounts. For simplicity the bulk data manager described in this paper supports operations on a single class of data. The same principles apply for operations on a database containing multiple classes of data. The design of a

more realistic functional database that supports multiple classes, views, security, alternative data structures and two data models is given in [Tri89].

In most existing languages only certain types of data may be permanently stored. Much of the effort in writing programs that manipulate permanent data is expended in unpacking the data into a form suitable for the computation and then repacking it for storage afterwards. The idea behind *persistent* programming languages is to allow entities of any type to be permanently stored. The length of time that an entity exists, or its persistence, is independent of its type.

In a persistent environment a class can be represented as a data structure that persists for some time. Because of their size such structures are termed bulk data structures. Operations that do not modify a bulk data structures, for example looking up a value, can be implemented efficiently in a functional language. However, modifications to a data structure must be non-destructive in a pure functional language, i.e. a new version of the structure must be constructed. At first glance it appears to be prohibitively expensive to create a new version of a bulk data structure every time it is modified.

3.2 Trees

A new version of a tree can be cheaply constructed. For simplicity the prototype data manager uses a binary tree. In a more realistic database a B-tree would be used. The distinction between binary and B-trees does not affect the efficiency described in the next Section. A class can be viewed as a collection of entities and there may be a key function that, given an entity, will return its key value. If et and kt are the entity and key types then an abstract datatype bdt, for a tree can be written

$$bdt = Node\ \ bdt\ \ kt\ \ bdt \mid Entity\ \ et$$

That is, every element of the tree is either an internal node with a left sub-tree, a key and a right sub-tree, or a leaf containing an entity. Using this definition, a function to lookup an entity can be written as follows. If the lookup succeeds the result returned is the required entity tagged Ok. If the entity does not exist, an $Error$ is reported.

$$\begin{aligned} lookup\ \ k'\ \ (Entity\ \ e) &= Ok\ \ e, \textbf{ if } key\ \ e = k' \\ &= Error, \textbf{ otherwise} \end{aligned}$$

$$\begin{aligned} lookup\ \ k'\ \ (Node\ \ lt\ \ k\ \ rt) &= lookup\ \ k'\ \ lt, \textbf{ if } k' \leq k \\ &= lookup\ \ k'\ \ rt, \textbf{ otherwise} \end{aligned}$$

A function to update an entity is similar except that, in addition to producing an output message, a new version of the tree is returned.

$$\begin{aligned} update\ \ e'\ \ (Entity\ \ e) &= (Ok\ \ e,\ \ Entity\ \ e'), \textbf{ if } key\ \ e = key\ \ e' \\ &= (Error,\ \ Entity\ \ e), \textbf{ otherwise} \end{aligned}$$

$$
\begin{aligned}
update\ e'\ (Node\ lt\ k\ rt) &= (m, Node\ lt'\ k\ rt), \textbf{ if } key\ e' \leq k \\
&= (m, Node\ lt\ k\ rt'), \textbf{ otherwise} \\
&\quad \textbf{where} \\
&\quad\quad (m, lt') = update\ e'\ lt \\
&\quad\quad (m, rt') = update\ e'\ rt
\end{aligned}
$$

The database-tree used to produce the results reported in the following Sections contains 2^{20} nodes. The database is lazily created by demanding the value of a database node. Because of GRIP's large primary memory (up to 128Mb), we assume initially that the entire database resides in primary memory. Results already obtained using simulated disk accesses [Tri90] may be subsequently verified on GRIP.

3.3 Efficiency

Let us assume that the tree contains n entities and is balanced. In this case its depth is $\log n$ and hence the update function only requires to construct $\log n$ new nodes to create a new version of such a tree. This is because any unchanged nodes are shared between the old and the new versions and thus a new *path* through the tree is all that need be constructed. This is best illustrated by the figure overleaf, that shows a tree which has been updated to associate a value of 3 with x.

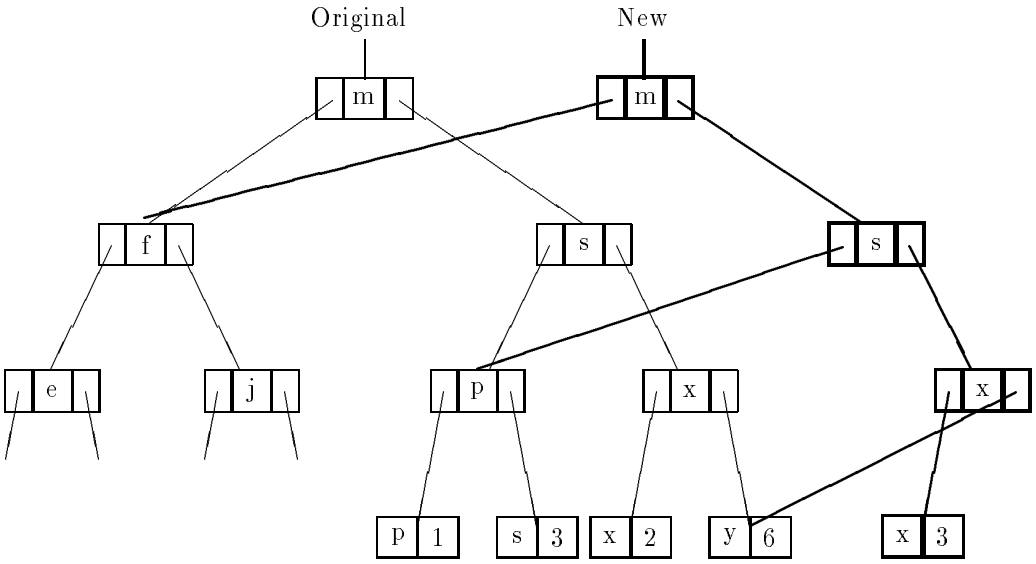
A time complexity of $\log n$ is the same as an imperative tree update. The non-destructive update has a larger constant factor, however, as the new nodes must be created and some unchanged information copied into them. The functional update can be made more efficient using a reference count optimisation [Tri89], but GRIP does not support this optimisation at present. However, when non-destructive update is used, a copy of the tree can be kept cheaply because the nodes common to the old and new versions are shared i.e. only a differential between the versions is required. This technique is similar to shadow paging [HG83] except that logical values, i.e. tree nodes, are being shadowed rather than fixed-sized pages.

As described in [AFHLT87, Tri89], the cheap shadow versions have several uses. They make undoing an aborted transaction easy: the original database is simply reinstated. Section 5 demonstrates how the multiple versions permit an unusual degree of concurrency between transactions. It has also been noted, but not thoroughly investigated, that this versioning technique could provide an elegant checkpointing recovery mechanism if used in conjunction with a stable store [Tri89].

3.4 Data Manager

The database-tree is controlled by a manager function that processes a stream of transactions to produce a stream of responses.

Original and New Trees



Transaction Functions

A transaction is a function that takes the database as an argument and returns some output and a new version of the database as a result. Let us call this type, $bdt \rightarrow (output \times bdt), txt$. Transactions are built out of tree manipulating operations such as *lookup* and *update*. In the bank database, two functions that prove useful are *isok*, that determines whether an operation succeeded, and *dep* that increments the balance of an account.

$$isok\ (Ok\ e) = True$$
$$isok\ out = False$$

$$dep\ (Ok\ Entity\ ano\ bal\ crl\ class)\ n\ =\ Entity\ ano\ (bal + n)\ crl\ class$$

Using *isok* and *dep*, a transaction to deposit a sum of money in a bank account can be written as follows.

$$\begin{aligned} deposit\ a\ n\ d &= update\ (dep\ m\ n)\ d, \textbf{ if } (isok\ m) \\ &= (Error, d), \textbf{ otherwise} \\ &\quad \textbf{where} \\ &\quad\quad m = lookup\ a\ d \end{aligned}$$

The arguments the *deposit* function takes are an account number a, a sum of money n and a database d. If the *lookup* fails to locate the account an error message and the original database are returned. If the *lookup* succeeds, the result of the function is the result of updating the

account. The update replaces the existing account entity with an identical entity, except that the balance has been incremented by the specified sum. Note that *deposit* is of the correct type for a transaction-function when it is partially applied to an account number and a sum of money, i.e. $deposit\ a\ n$ has type $bdt \rightarrow (output \times bdt)$.

The *deposit* transaction has a common transaction form: some operations are performed on the database and if they succeed the transaction commits, i.e. returns the updated database. If the operations fail, the transaction aborts and returns an unchanged database. Transactions that may commit or abort are termed *total*.

Manager

The database manager is a stream processing function. It consumes a lazy list, or stream, of transaction-functions and produces a stream of output. That is, the manager has type $bdt \rightarrow [txt] \rightarrow [output]$. A simple version can be written as follows.

$$\begin{aligned} manager\ d\ (f : fs) = {} & out : manager\ d'\ fs \\ & \textbf{where} \\ & \quad (out, d') = f\ d \end{aligned}$$

The first transaction f in the input stream is applied to the database and a pair is returned as the result. The output component of the pair is placed in the output stream. The updated database, d', is given as the first argument to the recursive call to the manager. Because the manager retains the modified database produced by each transaction it has an evolving state. The manager can be made available to many users simultaneously using techniques developed for functional operating systems [Hen82].

3.5 Concurrent Transactions

Total Transactions

Concurrency can be introduced between transactions by making the list constructor in the manager eager. A typical eager list constructor sparks a thread to evaluate the head of the list and continues to evaluate the tail of the list. The thread evaluating the head of the list, *out*, does so by applying the first transaction-function to the database, $f\ d$. The thread evaluating the tail of the list applies the manager to the updated database and the remaining transaction-functions, $manager\ d'\ fs$. The recursive call to the manager itself contains an eager list constructor that will spark a new thread to evaluate the second transaction and continue to apply the manager to the remaining transactions.

An eager manager provides ample concurrency between individual lookups and updates on the hypothetical machine [Tri90]. However a total transaction, i.e. one that may commit or abort, can seriously restrict concurrency. This is because neither the original nor the updated database can be returned until the decision whether to abort or not has been taken. Therefore no other transaction may access any other part of the database until this decision has been made.

Friedman-and-Wise If

Essentially total transactions have the form,

$$if \text{ predicate db } then \text{ transform db } else \text{ db.}$$

In most cases the bulk of the database will be the same whether or not the transaction commits. This common, or unchanged, part of the database will be returned whatever the result of the commit decision. If there were some way of returning the common part early then concurrency would be greatly increased. Transactions that only depend on unchanged data can begin and possibly even complete without waiting for the preceding total transaction to commit or abort.

The common parts of the database can be returned early using *fwif*, a variant of the conditional statement proposed by Friedman and Wise [FW78]. A more complete description of *fwif* and its implementation in the hypothetical machine can be found in [Tri89, Ake91]. To define the semantics of *fwif* let us view every data value as a constructor and a sequence of constructed values. Every member of an unstructured type is a zero-arity constructor — the sequence of constructed values is empty. Using C to denote a constructor, the semantics can be given by the following reduction rules.

$$\begin{array}{l} fwif\ True\ x\ y \Rightarrow x \\ fwif\ False\ x\ y \Rightarrow y \\ fwif\ p\ (C\ x_0 \ldots x_n)\ (C\ y_0 \ldots y_n) \Rightarrow C\ (fwif\ p\ x_0\ y_0) \ldots (fwif\ p\ x_n\ y_n) \end{array}$$

The third rule of *fwif* represents a family of rules, one for each constructor in the language. The third rule distributes *fwif* inside identical constructors, allowing the identical *then* and *else* constructor, C, to be returned before the predicate is evaluated. As an example, consider a conditional that selects between two lists that have 1 as the first element. Note how the first element of the list becomes available.

$$\begin{array}{ll} & fwif\ p\ (cons\ 1\ xs)\ (cons\ 1\ ys) \\ = & \{fwif\ 3\} \\ & cons\ (fwif\ p\ 1\ 1)\ (fwif\ p\ xs\ ys) \\ = & \{fwif\ 3\} \\ & cons\ 1\ (fwif\ p\ xs\ ys) \end{array}$$

To implement *fwif* the predicate and the two conditional branches are evaluated concurrently. The values of the conditional branches are compared and common parts are returned. When a part is found not to be common to both branches the evaluation of those branches ceases. Once the predicate is evaluated, the chosen branch is returned and the evaluation of the other is cancelled. This strategy amounts to speculative parallelism, the conditional branches being evaluated in the hope that parts of them will be identical.

4 Performance Measurements

In this section we present performance results gained from running database transactions on both GRIP and the hypothetical machine. The GRIP results are discussed in some detail, including investigation of the architectural implications of our results. In contrast, the hypothetical machine results are merely summarised in order to provide a comparison with an ideal machine.

We present results for lookup and update transactions independently. Our results are for the 2^{20} node database described earlier. *fwif* is used for the update transactions only.

4.1 Read-only Transactions

4.1.1 GRIP Results

Figure 1 shows the activity profiles for each of 6 PEs running 80 database lookups in 8 transactions. A typical lookup transaction is given in Appendix A.1. The GRIP database manager sparks a thread to evaluate each transaction in parallel.

In these results, a inverse relationship between reduction and communication time can be clearly observed. Generally the PEs are busy either computing or communicating. Reading and flushing (not separated in these plots) occur in roughly equal proportions throughout this run.

Figure 2 shows the activity and spark profiles for the same example aggregated over all PEs. The spark profile indicates that all sparks occurred early in the run. It also shows threads resuming after being blocked early in the run (the blocking is not shown). This blocking is a consequence of our distribution strategy, where a lookup transaction may not commence before the result of the previous transaction is known. Because the new, unchanged database is returned immediately from the lookups, the blocking period is always short.

The overall communication time is slightly higher than the overall reduction time (38% v. 37%). This indicates that few global nodes have been successfully cached. Given the random nature of the lookups and the fact that we are using a lazy caching algorithm, this is perhaps unsurprising. An eager caching scheme, where we prefetched graph nodes from global memory, might reduce PE communication overhead, as might a scheme which made better use of previously cached nodes.

The PEs are idle for 25% of the time. Idle time mainly occurs at the end of the run, when we are flushing the transaction pipeline. All PEs apart from PE 14.1 (which is running the transaction manager as the initial thread) are also partly idle at the start of the run. This presumably reflects the threads blocking on the root of the database, as noted earlier. The only significant idle time not mentioned so far occurs 75 ticks into PE 14.1's execution. This corresponds to the PE blocking, then resuming a transaction.

A rough estimate of absolute performance can made if the percentage reduction time is multiplied by the PEs in use. A single sequential processor performing no garbage collection or IO would have an absolute performance of 100%. For this example, our absolute performance is 222%. That is, we have definitely gained by using parallel evaluation in this case.

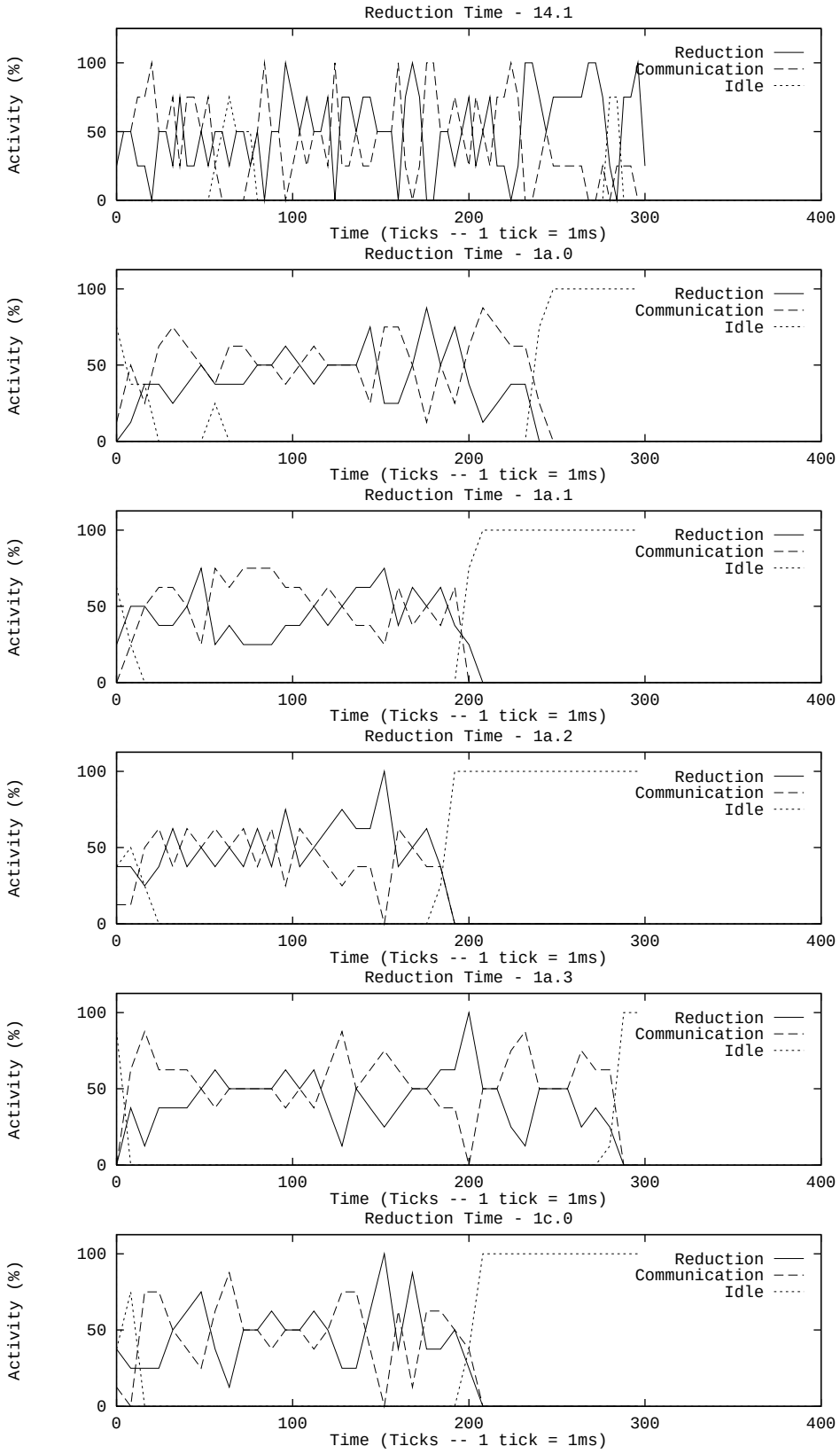


Figure 1: PE Activity Profiles: 80 lookups

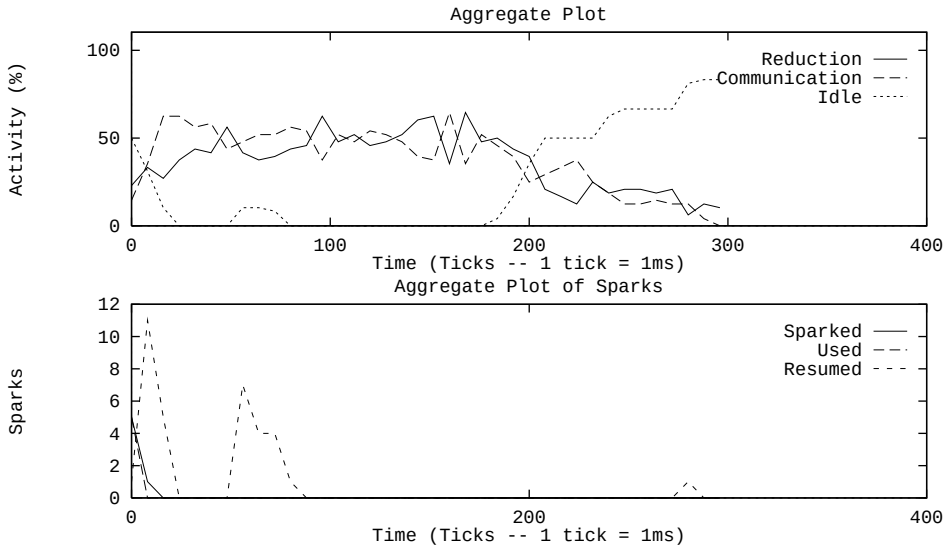


Figure 2: Aggregate Activity and Sparking Profiles: 80 lookups

It is interesting to observe that threads are only created at the start of execution. That is, all transactions are sparked early in the run. Each PE obtains one thread and sparks one new thread (corresponding to the tail of the transactions). Because there are 8 transactions, but only 6 PEs, there are 3 "spare" threads (one of which is trivial). These threads are obtained by those PEs which block first, and are held "in reserve".

4.1.2 Results from the Hypothetical Machine

The hypothetical machine can support the GRIP strategy of sparking a single thread for each lookup transaction. The active thread graph for evaluation of the 8 read-only transactions is given in Figure 3. The idealised hypothetical machine achieves an absolute speed-up factor of 6.67 over the sequential execution. This is considerably better than GRIP achieves, reflecting the shared memory and low thread-sparking cost.

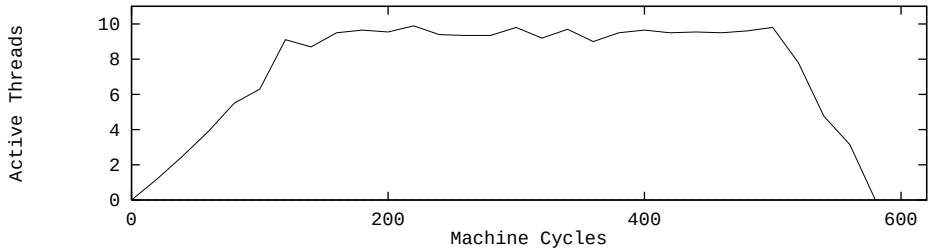


Figure 3: Simulator Results: 80 Lookups

There is some similarity between the numbers of active threads in the two machines. The hypothetical machine reaches a stable state where it is utilising approximately 9 processors. These represent the manager and the 8 read-transactions. The GRIP machine usefully employs 6 processors. We believe that GRIP could utilise more processors to evaluate a larger number of read-only transactions.

Because threads are so cheap on the hypothetical machine, *fwif* can be used to spark a thread for each lookup within the transactions. So much parallelism is generated by the program using *fwif* that 8 read-only transactions are not sufficient to reach a stable state. Hence Figure 4 plots the active thread graph for both a sequence of 8 read-only transactions (dotted line) and for a sequence of 32 read-only transactions (solid line). Assuming that thread sparking is so cheap is probably not reasonable if the hypothetical results are to be compared with GRIP results.

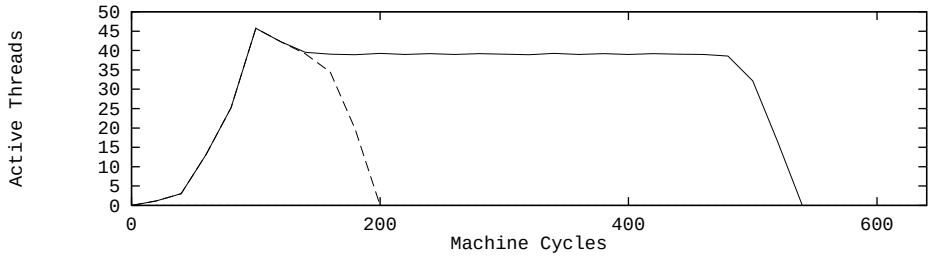


Figure 4: Simulator Results: Unrestricted Parallelism

4.2 Update Transactions

4.2.1 GRIP Results

To test the performance of the parallel database under more realistic conditions, we have implemented two transaction mixes which include database updates, insertions and deletions as well as lookups. We use a variant of *fwif* to enhance inter-transaction parallelism, as described above. LML source code for a typical update transaction is given in Appendix A.3. The first mix is a deliberately poor example, with many data interdependencies between transactions. The second mix aims to combine operations on related parts of the database, so that advantage can be taken of our cached copies of the database spine. This is realistic if transactions may be rearranged before processing.

For comparison, a typical update transaction in the first mix might update the records with keys 508440, 3420, 757150, 256330, 1001750, that is records in different quadrants of the database. Only the top two levels of the database are shared between the updates. A typical transaction in the second mix might update records with keys 8440, 3420, 7150, 6330, 1750. Now updates are occurring in the same sector of the database, and the top 10 nodes of the cached tree may be reused for the second and subsequent updates. Data dependencies between transactions will also be reduced. The first mix contains 330 operations in 34 transactions; the second mix contains 480 operations in 48 transactions.

Figure 5 shows aggregate activity and spark profiles for the first mix of transactions running on 4 PEs. For clarity, we have separated the plots of reduction, communication and idle time. The startup and tail-off periods where the transaction pipeline is filling and being emptied, respectively, can be clearly seen.

In this example, communication time closely follows reduction time (overall, 30% of the total runtime each). Initially, similar times are spent flushing and reading graph. However, from tick 3, more time is spent reading than flushing and from tick 19, reading dominates flushing by a factor of 2. Clearly, the database is effectively distributed by tick 19, and most communication consists of creating cached copies of the database.

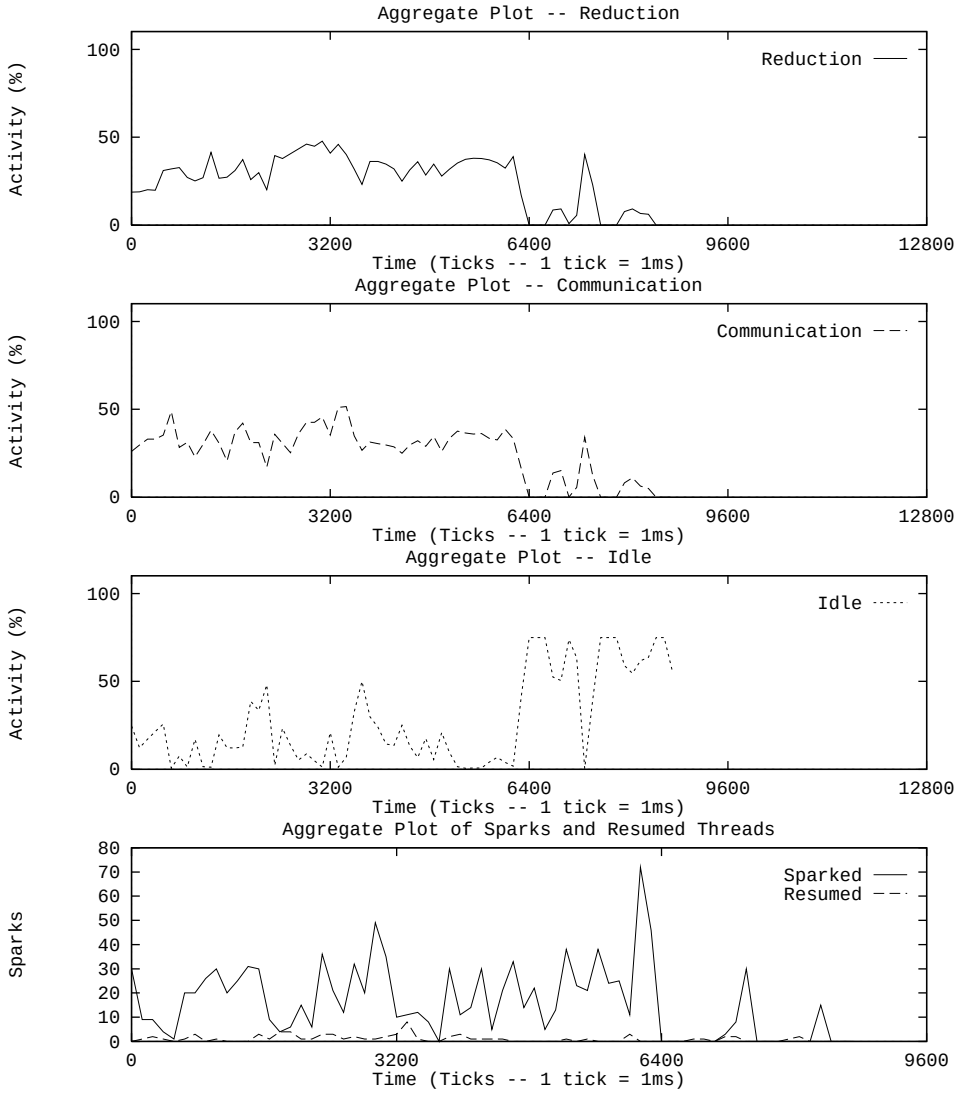


Figure 5: Aggregate Activity and Sparking Profiles: Mixed Transactions 1

Turning our attention to the sparking profiles, it is obvious that sparks are used to create threads almost immediately they are generated. The much higher sparking levels in this example reflect not only the additional transactions compared with the lookup example, but also the use of *fwif*, which in this version generates sparks to evaluate each level of the database in turn.

There is a small, but significant level of blocked thread resumption, corresponding to part of a database required by a later transaction being constructed by an earlier transaction. This will normally be the root of the database [Tri89]. Thread resumption thus corresponds to the release of database locks, where the data dependencies have been detected dynamically.

Garbage collection accounts for 7% of the total runtime. This is the highest percentage rate we have yet observed on GRIP. The relatively high cost is presumably due to high heap occupancy, perhaps because we are retaining blocked threads locally as long as possible, in order to avoid the costs associated with flushing these to global memory.

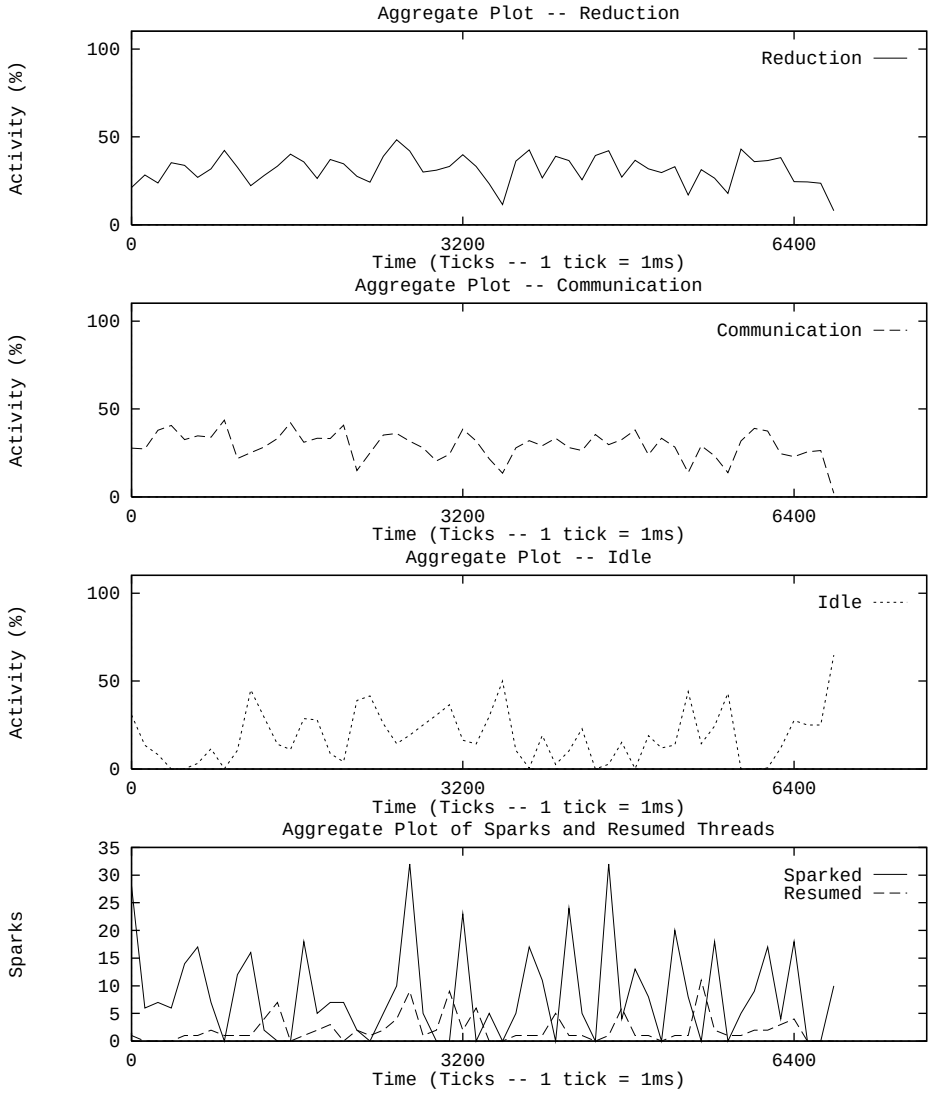


Figure 6: Aggregate Activity and Sparking Profiles: Mixed Transactions 2

Figure 6 shows the aggregate plots for the second (better) mix of transactions on 4 PEs. Once more, we have segregated the activity plots into communication, reduction and idle times.

Compared with the fully random mix discussed above, the startup and tail-off times are reduced considerably. Communication costs are also slightly reduced relative to reduction time (38% reduction, 35% communication). Garbage collection costs are also reduced to $< 5\%$ of the total runtime.

The sparking profile shows that far fewer sparks are generated than before – 252 sparks for 188 threads, compared with 1093 for 948 threads.

Given that we are evaluating more transactions in this example, in less elapsed time, it is clear that implicitly caching part of the database has improved performance.

The following table shows the absolute timings for this example on varying numbers of PEs, from 1 to 18. Up to 4 PEs, there is an obvious improvement for each PE added. Beyond 4 PEs, it is clear that the advantage from adding more processors diminishes rapidly.

PEs	Time(s)
1	13.248
2	7.456
3	3.885
4	3.570
5	3.424
6	3.398
8	3.005
10	3.338
12	3.141
18	3.325

For 8 PEs, we therefore achieve a maximum rate of 16 transactions processed per second. This seems a respectable rate for such a naive implementation. Significant performance improvements should result from tuning our system to reduce overheads and to take advantage of so far untapped processing resources.

4.2.2 Results from the Hypothetical Machine

The active thread graph for the hypothetical machine evaluation of the first update-transaction mix is given in Figure 7. For realism the thread-sparking strategy used is identical to that used in GRIP except where the *fwif* implementations and output drivers differ. The hypothetical machine achieves an absolute speed-up of a factor of 4.9 over the sequential evaluation. This ideal figure is considerably better than the GRIP absolute speed-up, though it is comparable with the relative speedup. The number of active threads in GRIP and in the hypothetical machine is similar. An average of 6.09 processors are active in the hypothetical machine while configuring GRIP to use more than 4 processors gains little improvement. The active thread graph for the second update-transaction mix is similar and is omitted.

5 Conclusion

We have implemented a parallel functional database for the GRIP declarative multiprocessor. Our database relies on the multiple versions produced by non-destructive updates to allow concurrent lookups and updates. Synchronisation between concurrent transactions is ensured by the normal thread synchronisation mechanism described above.

Our preliminary results show that we can achieve absolute speed-up over a sequential version of the database, even for the simple examples tried here.

We have yet not attempted to tune our results, beyond experimenting with improving the dynamic cache hit rate by combining operations on related parts of the database into single transactions. This experiment demonstrated that absolute performance could be improved by 33%, with a 10% reduction in communication cost, and a 29% reduction in garbage collection costs.

Comparison with other database systems suggests that our results are promising as early figures, especially if the communications overhead can be reduced and excessive data dependencies avoided. Tuning our system to take advantage of the unused processing capability should also improve our transaction throughput.

Communication overhead might be reduced by:

- Implementing eager memory prefetch. By initiating memory accesses for the children of a node when the node is demanded we can reduce latency and also reduce packet construction costs (the major component of our communication overheads).
- Combining dynamic data distribution with a good initial static placement. This is an open research area.

The relatively high cost for garbage collection (5%-7%) is probably due to retaining large numbers of blocked threads in the PEs rather than flushing these to global memory. Since each thread currently uses a 50Kbyte stack drawn from the heap. By experimenting with smaller chunks, and by being more eager to flush blocked threads we should be able to reduce GC costs. The remainder of the increase in GC costs must be due to maintaining our database in primary memory. Further experiments will be needed to determine what proportion of the GC cost should be apportioned to using such a large data structure.

References

[AFHLT87] Argo G, Fairbairn J, Hughes RJM, Launchbury EJ, and Trinder PW, "Implementing Functional Databases", *Proc Workshop on Database Programming Languages*, Roscoff, France (September 1987), pp. 87-103.

[Ake91] Akerholt G, "Extending A Parallel Functional Database", Senior Honours Project Report, Glasgow University Computer Science Dept., (1991), in preparation.

[Alv85] "Flagship Project — Alvey Proposal", Document Reference G0003 Issue 4, (May 1985).

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

[BW88] Bird RS and Wadler PL, *Introduction to Functional Programming*, Prentice Hall, (1988).

[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*, Portsmouth, New Hampshire, (1981).

[CGR89] Cox S, Glaser H, and Reeve M, "Compiling Functional Languages for the Transputer", *Proc Glasgow Workshop on Functional Programming*, Fraserburgh, Scotland, Springer Verlag, (August 1989).

[FW78] Friedman DP, and Wise DS, "A Note on Conditional Expressions", *Comm. ACM* 21(11), (November 1978).

[Hen82] Henderson P, "Purely Functional Operating Systems" in *Functional Programming and its Application*, Darlington J, Henderson P and Turner DA (Eds), Cambridge University Press, (1982).

[HG83] Hecht MS, and Gabbe JD, "Shadowed Management of Free Disk Pages with a Linked List", *ACM Transactions on Database Systems* 8(4), (December 1983), pp. 503-514.

[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*, Plasmeijer MJ (Ed), University of Nijmegen, (1990).

[HP91] Hammond K, and Peyton Jones SL, "Profiling Scheduling Strategies on the GRIP Parallel Reducer", submitted to *Journal of Parallel and Distributed Computing*, (1991).

[PCSH87] 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).

[Pey86] Peyton Jones SL, "Using Futurebus in a Fifth Generation Computer", *Microprocessors and Microsystems* 10(2), (March 1986), pp. 69-76.

[Pey87a] Peyton Jones SL, *The Implementation of Functional Programming Languages*, Prentice Hall, (1987).

[Pey87b] Peyton Jones SL, "FLIC - a Functional Language Intermediate Code", Department of Computer Science, University College, London, Internal Note 2048, (February 1987).

[PS89] Peyton Jones SL, and Salkild J, "The Spineless Tagless G-machine", *Proc FPCA 89*, London, MacQueen (Ed), Addison Wesley, (1989).

[Rob89] Robertson IB, "Hope$^+$ on Flagship", *Proc 1989 Glasgow Workshop on Functional Programming*, Fraserburgh, Scotland, Springer Verlag, (August 1989).

[Tri89] Trinder PW, *A Functional Database*, Oxford University D.Phil. Thesis, (December 1989).

[Tri90] Trinder PW, "Concurrent Data Manipulation in a Pure Functional Language", in *Proc 1990 Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Springer Verlag, (August 1990).

Appendix A: LML Sources

A.1 GRIP fwif

The operational definition of the *fwif* we have implemented for GRIP is shown below, in pseudo-LML. To define this function precisely, we need to use a number of pseudo-functions:

`boolval p` determines whether the predicate p is *True*, *False* or not yet evaluated (*Unknown*).

`eq p1 p2` determines whether *p1* and *p2* are identical local nodes or global pointers. Neither *p1* nor *p2* are evaluated.

`seq x y` evaluates x to weak head normal form, then evaluates and returns y.

`par x y` sparks x and continues execution of y.

C stands for an arbitrary n-ary constructor. All uses of C refer to the same constructor.

We assume that *fwif* is never applied to $\perp$. For the database example described here, this is always the case.

```
rec fwif p x y =
  case boolval p in
       True : x
    || False : y
    || Unknown :
         if eq x y then x
         else seq x (seq y (
           if eq x y then x
           else case x in
             C x1 ... xn :
               case y in
                 C y1 ... yn :
                        let xy1 = fwif p x1 y1
                        ...
                        and xyn = fwif p xn yn in
                        par xy1 (... (par xyn (C xy1 ... xyn)) ... )

                 || _ : if p then x else y
               end
           end))
  end
```

For the database example, we know that only one branch of the constructor (node) will have changed between transactions. In fact, in general, it is likely that only some subnodes of an constructor will vary between x and y, and we need only spark *fwif* calls on non-identical children. Although the *fwif* threads would terminate immediately in the case where the children were identical, it is better to avoid the overhead of creating such fine-grained threads if possible. We therefore test for identity before creating the recursive calls.

```
let xy1 = fwif p x1 y1 and xy1' =
...
and xyn = fwif p xn yn in
if not (eq x1 y1) then
par xy1 xy2'

par xy1 (... (par xyn (C xy1 ... xyn)) ... )
```

Although we would prefer to return the constructor node immediately, and then nominate one spark as the successor of the *fwif*, this is not possible with the operations we have specified so far. Further work will investigate a version of *fwif* implemented in this way.

A.2 Read-only Transaction

This appendix gives a sample read-only transaction in LML. The transaction comprises 10 lookups on the same database (that is there is no interdependency between lookups). Since the database is unmodified by lookup operations, this result may be ignored. The new, unchanged database may be returned before any lookup is evaluated.

```
lookups d =
    let (o1,_) = lookup   1000     d in
    ...
    let (o10,_) = lookup  2010     d in
        ((if (isok o1) & (isok o2) & (isok o3) & (isok o4) & (isok o5) &
           (isok o6) & (isok o7) & (isok o8) & (isok o9) & (isok o10) then
         o10 else error "dep" 0),d)
```

A.3 Update Transaction

This appendix shows a sample update transaction written in LML. The transaction comprises 5 lookups, each of which is followed by an update of the same database node. The update is thus dependent on its associated lookup, and successive lookup/update pairs are dependent on their predecessors.

```
updates n d =
    let (o1,_) = lookup   508440     d in
    let (o2,d1) = update (deprec o1 n)     d in
    ...
    let (o9,_) = lookup   1001750     d in
    let (o10,d5) = update (deprec o9 n)     d4 in
    let test = (isok o2) & (isok o4) & (isok o6) &
               (isok o8) & (isok o10) in
    let d5 = fwifdb test d5 d in

    par d5 ((if test then o10 else error "ups1" 0),d5)
```

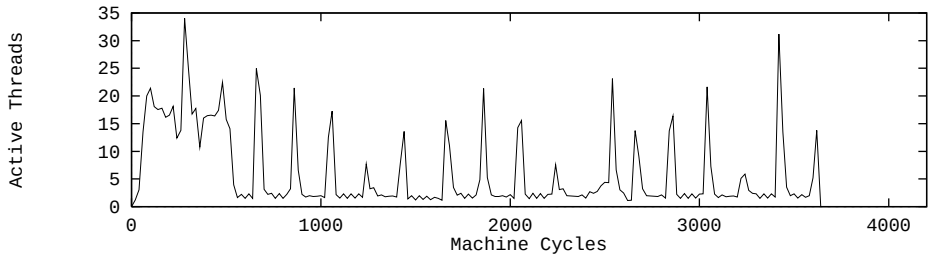


Figure 7: Simulator Results: Mixed Transactions 1