PROCESSING TRANSACTIONS ON GRIP, A PARALLEL GRAPH REDUCER

G. AKERHOLT, K. HAMMOND, S. PEYTON JONES AND P. TRINDER

To appear in Proc. PARLE '93, Munich, June 1993.

Abstract. The GRIP architecture allows efficient execution of functional programs on a multi-processor built from standard hardware components. State-of-the-art compilation techniques are combined with sophisticated runtime resource-control to give good parallel performance. This paper reports the results of running GRIP on an application which is apparently unsuited to the basic functional model: a database transaction manager incorporating updates as well as lookup transactions. The results obtained show good relative speedups for GRIP, with real performance advantages over the same application executing on sequential machines.

1. Introduction

GRIP is a parallel processor designed for fast, efficient execution of pure functional programs. Good sequential compiler technology is combined with parallel runtime support to give good real-time performance. Pure functional languages form an attractive basis for parallel implementation, if a safe evaluation strategy such as parallel graph reduction is used:

- The principle of referential transparency ensures that all cached copies of a given object will have the same value when evaluated, whether or not they are shared, and no matter how many times they are evaluated. Thus, there can be no cache-coherency problems in a parallel functional implementation.
- The semantics of a functional program remains the same whether it is executed sequentially or in parallel. Thus, a parallel functional program may be debugged on a sequential machine without affecting its result. There can be no unexpected non-determinism in a parallel functional program.
- There is no possibility of deadlock. Parallel functional programs have exactly the same termination properties as their sequential counterparts.
- Because there is no explicitly sequential evaluation order, it is easy to automatically partition a functional program for parallel execution.
- Automatic resource-control is much more straightforward, since there are no hidden dependencies between tasks.

A number of pragmatic issues remain, however, for example whether *good* partitions into tasks can be made without human intervention, or whether dynamic control decisions

This work is supported by the ESPRIT FIDE Project (BRA 3070), the SERC Bulk Data Type Constructors Project, the SERC GRASP Project and the Royal Society of Edinburgh. **Authors' address:** Computing Science Dept, Glasgow University, Glasgow, Scotland. **Email:** {akerholg, kh, simonpj,trinder}@dcs.glasgow.ac.uk

can be made sufficiently fast to allow scheduling of fine-grained parallelism on a machine such as GRIP. We have addressed some of these issues in earlier papers [HP92]. In this paper, we consider another important pragmatic issue: whether functional programs can be made to process large amounts of data in a manner which is competitive with imperative programs. We have chosen as our case-study a partial implementation of the well-known DebitCredit benchmark: a transaction-processing benchmark for databases, which involves updating the database. Given that sequential compilers for functional languages do not yet give performance which matches that of imperative languages, we do not expect our implementation to outperform a hand-coded imperative program for the same machine. We do, however, hope to obtain respectable performance compared with imperative implementations, and to obtain decent speedups from our parallel architecture. Choosing a widely-accepted benchmark allows tentative comparisons to be drawn with other architectures and models of computation.

While the results we obtain here apply principally to our novel GRIP architecture, there is some hope that the lessons learned here may also be of use to other parallel functional implementations, such as those for networks of transputers, or hypercubes. Although the GRIP model lessens the problems of locality through the use of a 2-level bus structure and fast heterogeneous communications hardware, the distinction between local and non-local memory accesses is still a crucial one.

The remainder of this paper is structured as follows. Section 2 describes the GRIP machine architecture. Section 3 describes the characteristics recorded during program execution on GRIP. Section 4 describes the DebitCredit-based application studied here. Section 5 gives the results gathered during the execution of the application. Section 6 concludes.

2. Machine Architecture

2.1. Overview. The GRIP architecture comprises a single bus-connected cluster of one to 20 printed circuit boards. A fully-populated board contains four processing elements (PEs) and one Intelligent Memory Unit (IMU), linked by a local bus. A fully-populated GRIP thus contains 80 PEs and 20 IMUs. The boards are connected using a fast packet-switched bus [Pey86], and the whole machine is attached to a Unix host using slower data links.

Each PE incorporates an MC68020 CPU, an MC68881 floating-point co-processor, and 1Mbyte of private memory which is not accessible by any other hardware component.

The IMUs collectively constitute the global address space. They each contain 1M words of 40 bit-wide static memory, 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 IMUs are the most innovative feature of the GRIP architecture, offering a fast implementation of low-level memory operations with great flexibility.

An internal bus was chosen specifically to make the locality issue less pressing. Communication is handled by sophisticated Bus Interface Processors (BIPs): one per board. Identical protocols are used for communication between remote components or those on the same board. Throughput and latency are essentially the same for both local and remote communication from functional programs [Mad91]. However, inter-component communication is still an order-of-magnitude slower than access to a PE's private memory. It is thus crucially important to minimise the number and frequency of remote accesses.

2.2. Graph reduction on 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 [Pey87]. Our intention is that, provided a thread does not refer to remote graph nodes, it should be executed exactly as a compiled program would be on a sequential machine.

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*, held in dynamic heap memory. Each closure consists of a pointer to its *code*, together with zero or more *free-variable fields*. Closures in *(weak head) normal form* require no further evaluation, hence their code is usually just a return instruction[1]. Other closures represent unevaluated expressions, whose code will reduce the closure to its normal form. A closure is evaluated (or *entered*) by jumping to its code. A register records the current closure for update purposes, or for access to the free variables. 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. In a parallel graph reducer, there will typically be many threads which could be executed. Idle PEs fetch new threads from this (distributed) pool of threads. A single PE may execute one thread at a time, or may multi-task between a number of threads.

Initially there is only one thread, representing the result of the program. When a thread encounters a closure whose value will be required in the future, it has the option of recording the closure for (possible) execution by other PEs. This is known as *sparking* the closure.

If the parent thread requires the value of the sparked closure while a 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*. If no PE has begun execution of the sparked closure when its value is required, the parent thread will evaluate the closure itself. Consequently a thread can only become blocked if it requires a result which some other thread is evaluating [PS89, HP90]. This is the *evaluate-and-die* model of evaluation for parallel functional languages. It is related to some other models such as lazy task creation [MKH91].

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. The 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.

2.3. IMU Operations. The following range of operations is supported by our current IMU microcode:

- Variable-sized heap nodes may be allocated and initialised.
- Garbage collection of global nodes is performed autonomously by the IMUs. Termination is ensured using an algorithm proposed by Baker [Bak78].
- Each IMU maintains a pool of executable threads, which may be exported to idle PEs.
- The blocking/resumption model is supported for access to global nodes.

[1]Closures in weak head normal form are functions, or constructors. In contrast to true normal forms, their arguments may be unevaluated.

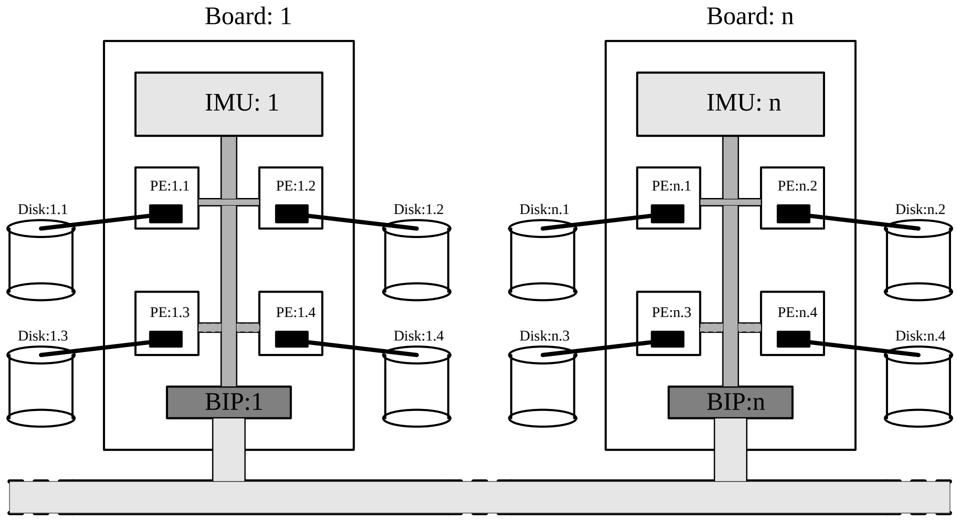


FIGURE 1. GRIP extended with disks and disk controllers

2.4. Additional Configuration. Database applications require that the underlying machine supports permanent storage, typically in the form of disks. The existing GRIP machine has a simple stream-based disk interface, which is clearly inadequate for such an application. Pragmatics aside, it would be easy to extend the GRIP architecture to include more sophisticated disk storage. For example, a disk controller and disk could be added to each PE. Any PE could then access the data on any disk by sending a suitable request to the controlling PE. This architecture is depicted in Figure 1.

To simulate this architecture on the existing machine, a special primitive operation, *delay*, is used to model disk accesses. *delay n a* introduces a timed delay of n milliseconds for the current thread. To model the effect of contention for shared resources, such as disks, delays are queued on the PE addressed by a, and are cumulative with any outstanding delays on that PE. We use a delay of 13ms for a disk read, and 14ms for a disk write, and assume a block size of 8K bytes, with a total capacity of 1G bytes per drive. These correspond to figures quoted for many small SCSI drives, e.g. Seagate ST41200N 1.2Gb (16.5ms) or Maxtor 1.7Gb (13ms).

3. Database Architecture

3.1. DebitCredit. The database application that has been implemented on GRIP is the processing part of the DebitCredit benchmark[Tpc89]. This section describes the significant features of this application. DebitCredit measures transaction processing capacity in a simple bank database. The full benchmark measures transactions passed over a network from a set of terminals and includes pricing information for the entire system. The application we describe only processes the transactions against the database: the so-called back-end processing. Network response times and equipment costs are not considered here.

The DebitCredit bank database comprises customer, teller, branch and history records. A single transaction is repeatedly executed against these records. The transaction adds an amount of money to an account (a negative amount is a withdrawal), the corresponding teller and branch records are similarly updated and a history record is generated. Various relationships exist between the records, for example the balance held at a branch should be the sum of all of the accounts at the branch. The benchmark specifies a set of atomicity, consistency, isolation and durability (ACID) tests. All except the durability test have been performed successfully for the program described here. Results are omitted for space reasons.

The essential metric measured in DebitCredit is the number of transactions processed in a second (tps). However, the database size does not remain constant as the tps rate increases. For each transaction-per-second the database must use 100,000 account records, 10 teller records and 1 branch record.

DebitCredit figures have been published for many machines. The figures reported in this paper are for only part of DebitCredit, involve simulated disk access and, as described in the next section, deviate from the specification in several respects. Hence they cannot be directly compared to a full implementation. However, for reference, the following figures are quoted by e.g. [Rob89, TPG88]: IBM 4381-P22, 22 TPS; DEC VAX 8830, 27 TPS; Tandem, 208 TPS.

3.2. Application Design.

3.2.1. *Persistent Functional Languages.* The transaction processor is designed using the principles first outlined in [AFHLT87] and prototyped in [AHPT91, Tri89], which assume the existence of a parallel persistent functional language. 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 values 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, or collection of 'similar' data items, can be represented as a data structure that persists for some time. Because of their large size, such structures are termed bulk data structures. Operations that do not modify bulk data structures, e.g. lookups, can be implemented efficiently in a functional language [HN91]. 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 and the original preserved. At first glance it seems to be prohibitively expensive to create a new version of a bulk data structure every time it is modified.

3.2.2. *Trees.* New versions of trees can be constructed cheaply, however. If *et* is the type of the data values at the leaves, and *kt* is the type of the keys, then a simplistic tree type can be written

$$bdt = Node\ bdt\ kt\ bdt \mid Tip\ et.$$

A function to update such a tree produces a new tree reflecting the update and a message reporting the success or failure of the operation.

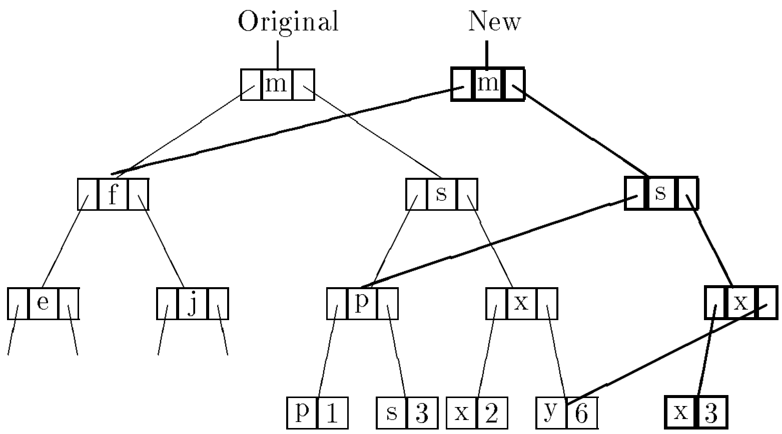
$$\begin{aligned} update\ e'\ (Tip\ e) &= (Ok\ e,\ Tip\ e'),\ \textbf{if}\ key\ e = key\ e' \\ &= (Error,\ Tip\ e),\ \textbf{otherwise} \end{aligned}$$

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

Let us assume that the tree contains n entities and is balanced. In this case its depth is proportional to $\log n$, hence the update function needs only to construct $\log n$ new nodes to create a new version of such a tree. Any unchanged nodes can be shared between the old and the new versions and thus a new *path* through the tree is all that need be constructed. The figure overleaf 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 reference counting [Tri89], but the GRIP software does not currently support this optimisation. 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 the differences between the versions are required. The uses of cheap multiple versions of the database are described in [AFHLT87, Tri89]. Destructive update is also likely to introduce unwanted sequential dependencies (as its name suggests, single-threading imposes sequential access in order to allow destructive update). This is highly undesirable for our parallel application.

The DebitCredit branch, teller and account classes are each represented as trees, while the history is simply a sequence. Because the branch and teller classes are small enough they are stored entirely in primary memory, as binary trees with data (a single 100 byte record) only at the leaves. The account tree is too large to reside in primary memory, hence we use a 2-3 tree (i.e. a B-tree of order 3).



A disk-block access is simulated for each leaf access, as described in Section 2.4. This corresponds to an ideal 'warm start' in a conventional database, i.e. all of the index is in memory and only the data is disk resident. Each DebitCredit account record is 100 bytes and hence 80 records are retrieved from an 8Kb disk-block.

We choose a low-order B-tree to minimise the construction time required for the root node. This reduces a potential throughput bottleneck [Tri89]. Future work may include experimenting with the order of the B-tree.

3.2.3. *Transaction Manager.* 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*. Two functions that prove useful to construct a simple example transaction are *isok*, which determines whether an operation succeeded, and *dep* which increments the balance of an account. The arguments to *dep* are a some of money to deposit n and an entity whose components are an account number ano, the current balance of the account bal, the credit limit for the account crl, and the type of the account $class$.

$$\begin{array}{l} isok\ \ (Ok\ \ e) = True \\ isok\ \ out = False \end{array}$$

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

A transaction to deposit a sum of money in a bank account can be written as follows.

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

The *deposit* function takes as its arguments 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 DebitCredit transaction, $dctrans$ which is used for performance analysis is much more complicated than *deposit*. Its definition is given in Appendix A.

Both *deposit* and $dctrans$ have 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 either commit or abort are termed *total*.

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.

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

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].

3.2.4. *Concurrent Transactions.* Concurrency can be introduced between transactions by making the manager eager. This allows the current transaction to be evaluated in parallel with the remaining transactions. The original task evaluates the current transaction. The new task applies the manager to the remaining transactions. This proceeds recursively.

Unfortunately, total transactions can seriously restrict concurrency. This is because neither the original nor the updated database can be returned until the commit/abort decision has been taken. Consequently, no other transaction may access any other part of the database until this decision has been made. 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 a simulated parallel graph reducer can be found in [Tri89]. 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, e.g. 1, 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.

$$fwif\ True\ x\ y \Rightarrow x$$
$$fwif\ False\ x\ y \Rightarrow y$$
$$fwif\ p\ (C\ x_0 \ldots x_n)\ (C\ y_0 \ldots y_n) \Rightarrow C\ (fwif\ p\ x_0\ y_0) \ldots (fwif\ p\ x_n\ y_n)$$

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.

The problems of speculative parallelism are well known. For example speculative tasks may consume resources and hence prevent more important tasks from completing. They, and any child tasks, may also be hard to kill if they are not required, or fail to terminate. Fortunately in the DebitCredit application the tasks being sparked by *fwif* evaluate functions like *update* which are relatively small, spark no additional tasks and are guaranteed to terminate as they traverse a finite data structure. Since most of the database is unchanged between transactions, the speculative work is likely to be used.

4. Results

4.1. Performance on Parallel and Sequential Machines. The first set of results compares the absolute execution times of a fixed program running on a varying number of GRIP PEs with those for the same program executing on two common sequential machines. This program processes 400 transactions on a database configured for 50 DebitCredit TPS. Figure 2 plots the execution times for a 4-IMU GRIP (20Mb "slow" global heap) with between 2 and 15 PEs (this was the largest stable configuration at the time these results were obtained). Each PE has 600K available heap (the remaining 400K static RAM is occupied by program code, the operating system, and static data). Due to the size of the application, it could not be executed on a single GRIP PE. Offloading the in-memory index to global memory would bias the performance results, and give unrealistic superlinear speedups, which we wished to avoid.

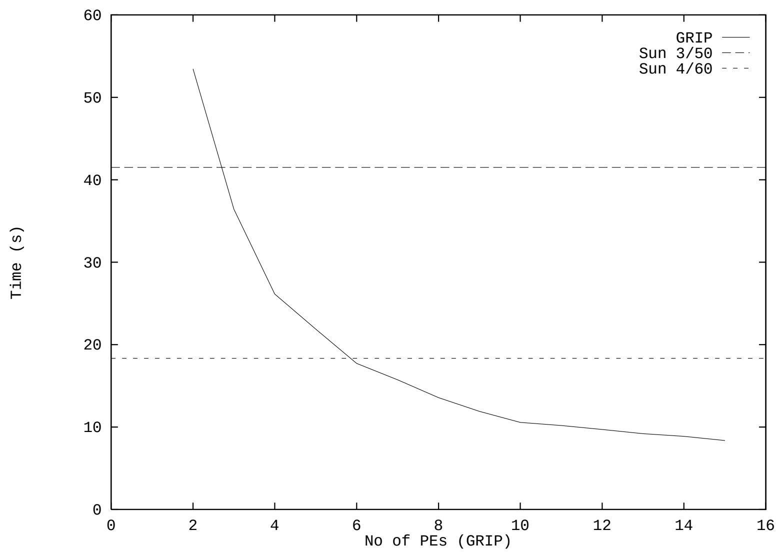


FIGURE 2. Execution Time Profile: 400 Transactions, 50 TPS Database

The same figure also shows the execution times for an identical program executing on a Sun 3/50 (Motorola MC68020) and a Sun 4/60 (Sun Sparc). The Sun 3/50 uses the same processor at the same speed as GRIP; the Sun 4/60 is a commonly used modern machine. An 8M heap was used for the sequential machines: this gave the best overall time performance in both cases. The same compiler was used for all three machines. Disk accesses were simulated for GRIP using interrupt-timed delays, as described above. For the sequential versions, sequential disk access was simulated using count-down loops of an appropriate duration. The implementation of delays is the only difference between the programs. Results are averaged across 10 runs in each case.

A direct architectural comparison can be made between GRIP and the Sun 3/50. They use the same microprocessor (16MHz Motorola MC68020) to execute an essentially identical source program compiled by the same compiler. The machines differ in their memory architectures, in their communications sub-systems, and in their virtual disk architectures, as described above. Absolute speedups over the Sun 3/50 are obtained with 3 or more PEs. Not all the overhead is due to communications and context-switching costs, however: a significant fraction of this overhead is caused by the relatively small local memory available to each GRIP PE (for example, decreasing the Sun 3/50 heap to 2Mb halves its overall performance).

The raw integer performance (given by SPECint89) of the Sparc-1 chip used in the Sun 4/60 is roughly 3 to 4 times that of the Motorola 68020. For this application, however, disk performance is at least as important as that of the processor. Consequently, the overall performance of the 4/60 is only twice that of the 3/50, in spite of using a RISC chip. Hence a GRIP with 6 or more PEs outperforms the Sun 4/60, and a 15-PE GRIP is more than twice as fast as a Sun 4/60. This is a primarily a consequence of our use of concurrency to exploit additional disks in the parallel machine.

To summarise, for this program, a 15-PE GRIP delivers good real-time performance compared with some common sequential machines.

4.2. Relative Speedup. The second set of results investigates the relative speedup as the number of processors is increased from 2 to 15. Figure 3 plots the same data as Figure 2, but in terms of the speedup relative to the two-PE case (the single PE data-point was unobtainable, but extrapolating from our data suggests a single PE would be roughly half

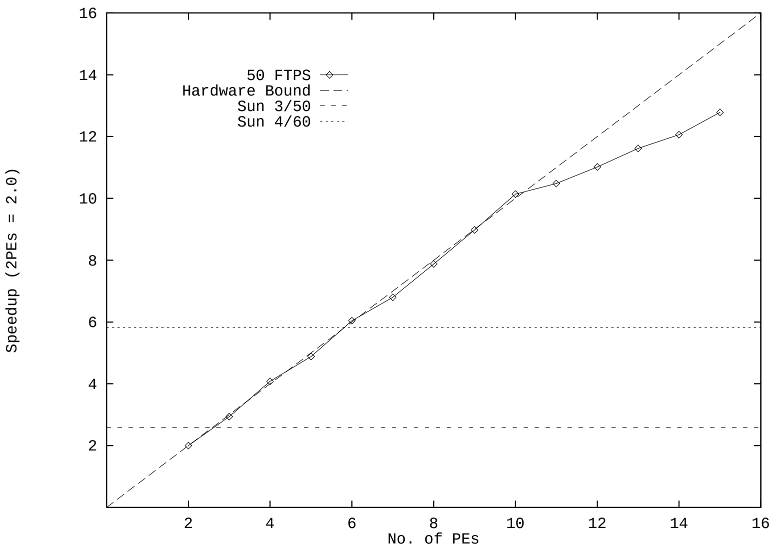


FIGURE 3. Speedup Graph: 400 transactions, 50 TPS Database

the speed of a 2-PE system). That is, the program measured executes 400 transactions against a database configured for 50 DebitCredit TPS. We observe that the speedup is linear until 10 PEs are in use and degrades thereafter.

This degradation in speedup occurs as the software bound on parallelism is approached. The bound for a functional transaction processor has been shown to be the ratio between the time required to construct the root and make it available to other processors and the time required to process the transaction [Tri89]. Hence we would expect that, if the transactions became shorter, the software bound would be reached sooner.

The length of DebitCredit transactions is easily adjusted. Recall that the time to execute a transaction is proportional to the log of the size of the database. Furthermore, the benchmark specifies that the size of the database increases in proportion to the number of TPS. The largest database studied here is the 50 TPS database from the previous section, selected because a 15-PE GRIP achieves 48 FTPS (400 transactions in 8.36 seconds). The smallest database we consider is a 15 TPS database, chosen because a 2-PE GRIP achieves 14 FTPS. A good intermediate point is a 35 TPS database, chosen because an 8-PE GRIP achieves 36 FTPS.

Figure 4 plots the speedup curves for the 400 transaction program executed on 15 TPS, 35 TPS and 50 TPS databases respectively. As predicted, a program with shorter transactions (and hence a smaller database) reaches the software bound earlier. We note that considerable improvement is still obtained after the speedup becomes non-linear. In fact, none of the programs have actually reached a limit on speedup. This suggests that the DebitCredit execution time could be further reduced by increasing the number of processors beyond 15.

5. CONCLUSION

We have run a large data-intensive application on the parallel graph-reducer GRIP. Our application is written in the pure non-strict functional language, Haskell. It exploits the data-dependencies implicit in a functional program to provide inter-transaction concurrency and locking. This represents the first attempt that we are aware of to consider the problems of concurrency in a functional transaction processor, in the presence of update as well as lookup transactions. Our model allows the exploitation of concurrent hardware

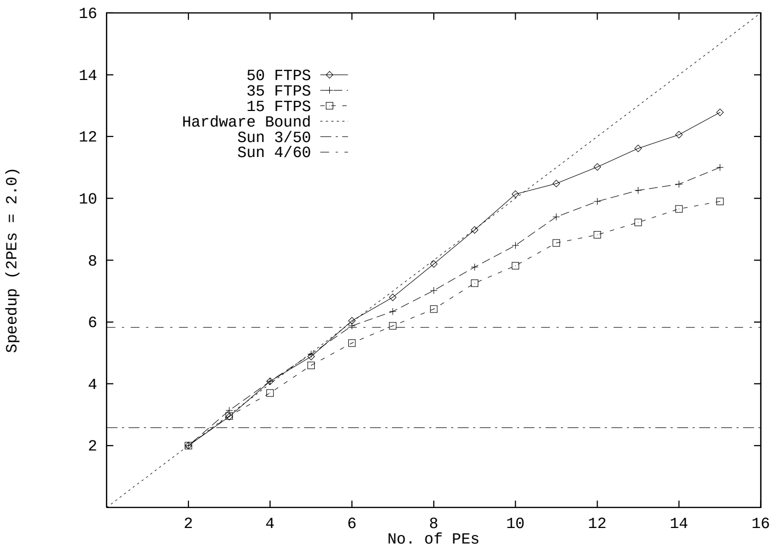


FIGURE 4. Parallel Speedup Graphs for Varying TPS

through the use of *fwif*: a primitive which allows early return of common parts of a data structure in the context of conditional expressions.

Our results show a clear improvement from the use of a parallel machine compared with the same application running on two popular sequential machines. We also obtain near-linear relative speedups between 2 and 10 PE on a 50-TPS database. Our GRIP results could be tentatively compared with those for the the full DebitCredit benchmark on a large sequential machine.

There are several aspects which would repay further investigation. These include:

(1) One PE, with varying numbers of "virtual disks" to measure the potential concurrency gain for a sequential version exploiting *fwif*.
(2) Other architectures, for example a shared-memory machine such as the Sequent Symmetry, using the Chalmers $<\nu$,G$>$-machine compiler [AJ89].
(3) Eager pre-fetch of normal-form packets from the IMUs.
(4) Local sparking [HP92] to reduce communication overhead.
(5) An improved implementation of *fwif* with indirection-chaining via the IMUs.

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.

[AHPT91] Akerholt G, Hammond K, Peyton Jones SL, and Trinder P, "A Parallel Functional Database On GRIP", *Glasgow Workshop on Functional Programming*, Portree, Scotland (August 1991).

[AJ89] Augustsson L and Johnsson T, "Parallel graph reduction with the $<\nu$,G$>$-machine", *Proc IFIP Conference on Functional Programming Languages and Computer Architecture, London, (September 1989).*

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

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

[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).*

[HP92] *Hammond K, and Peyton Jones SL, "Profiling Scheduling Strategies on the GRIP Parallel Reducer",* Proc 4th Intl Workshop on Parallel Implementation of Functional Languages, *Kuchen H and Loogen R (Eds), RWTH, Aachen, (1992).*

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

[HN91] *Heytens, M.L. and Nikhil R.S. "List Comprehensions in AGNA, a Parallel Persistent Object System"* Proc FPCA 91, *Cambridge, Mass. (1991).*

[Mad91] *Madden, P.J. "The Hardware Performance of the GRIP Multiprocessor"* MSc Thesis, *Glasgow University, (1991).*

[MKH91] *Mohr E, Kranz DA and Halstead RH, "Lazy task creation - a technique for increasing the granularity of parallel programs"* IEEE Transactions on Parallel and Distributed Systems, *2(3), (July 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.*

[Pey87] *Peyton Jones SL, The Implementation of Functional Programming Languages, Prentice Hall, (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).*

[Tpc89] *Transaction Processing Performance Council (TPC), "TPC BENCHMARK A, Draft 6-pr Proposed Standard",* Administered by ITOM INternational Co, POB 1450, Los Altos, CA 94023, USA, *(August 1989).*

[TPG88] *The Tandem Performance Group, "A Benchmark of NonStop SQL on the DebitCredit Transaction",* Tandem Computers Inc., 19333 Vallco Pky., Cupertino, CA. 95014, *(1988).*

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

Appendix A DebitCredit Program

```
module Types where              ------------- Type Definitions  -------------

data Fill =     Fill_Bra Int Int Int | Fill_Tel Int Int
data Entity =   Branch Int Int Fill | Teller Int Int Int Fill
data Tree =     Tip Entity | Tip_Acc Int Int |  Node1 Tree Int Tree |
                   Node2 Tree Int Tree Int Tree
data BOOL =     FALSE | TRUE | UNKNOWN


module Main where               -------------    Main Module    -------------

infixr `seq`, `par`

import FWIF (fwifdb)            -- fwifdb :: Bool -> Tree -> Tree -> Tree
import Types(Fill,Entity,Tree)
import Delay(delaya)            -- delaya :: Int -> Int -> Int

data Histrt     =       His      Int      Int      Int      Int      Int
data Dbt        =       Root     Tree     Tree     Tree     [Histrt]
data Msgt       =       Ok       Int
                |       Error    Int
```

```
isok :: (Msgt, tree) -> Bool
isok (Ok k, t) =     True
isok (Error k, t) =  False

----------------   ... Code to build the database trees ...   ---------------

replace :: Int -> Int -> Tree -> (Msgt, Tree)

replace key d nd@(Tip_Acc aid nrec) =
        if not is_error && key >= aid && key <= aid+nrec then
             write_disk 14 (read_disk 13 (Ok key, (Tip_Acc aid nrec)))
        else
             read_disk 13 (Error key, nd)
        where
                read_disk d cont  =  read_delay d `seq` cont
                        where read_delay n =   delaya n key
                write_disk d cont =  write_delay d `seq` cont
                        where write_delay n =  delaya n key

                is_error = (key `div` 10) `mod`  20 == 0

replace key d (Node1 lt k rt) =
        if key > k then     (msg_r, Node1 lt k new_rt)
        else                (msg_l, Node1 new_lt k rt)

        where               (msg_r, new_rt) = replace key d rt
                            (msg_l, new_lt) = replace key d lt

------------------   ... Code for Tip and Node2 cases ...   -----------------

{- The following function represents one DebitCredit transaction.         -}

dctrans :: Int -> Int -> Int -> Int-> Transaction

dctrans aid bid tid delta db =
        ret_acc `par` (ret_bra `par` (ret_tel `par`
        (Ok (acct a_result), Root ret_acc ret_bra ret_tel ret_his )))
        where
                (Root acc bra tel his) = db

                a_result = replace aid delta acc
                b_result = replace bid delta bra
                t_result = replace tid delta tel
                h_result = (Ok 0, His aid bid tid delta 0:his)

                p = isok a_result && isok b_result && isok t_result

                ret_acc = fwifdb p (snd a_result) acc
                ret_bra = fwifdb p (snd b_result) bra
                ret_tel = fwifdb p (snd t_result) tel
                ret_his = if p then snd h_result else his

                acct (Ok aid,_) =       aid
                acct (Error aid,_) =    aid

---------   ... Code to generate a list of random transactions ...   --------
```

```
manager :: a -> [a -> (b,a)] -> [b]

manager d (f:fs) =  ms `par` ml
        where   fd = f d
                (m,d') = fd
                ms = manager d' fs
                ml = m `seq` (m:ms)
manager d [] = []

----------   ... Code to calculate a checksum of results etc ...   ----------

main _ = db `seq` txs `seq` show (checksum result)
        where   db = builddb cur_tps cur_nrec
                txs = randtxs cur_ntxs cur_tps
                result = (manager db txs)
```

Appendix B Outline of FWIF Implementation

The following is a listing of the *fwifdb* function, or Friedmann and Wise if [FW78] for the database type *Dbt*, in Haskell. The **eq** pseudo-function tests whether its arguments are pointer identical. The **boolval** pseudo-function returns **TRUE** if its argument is **True**, **FALSE** if its argument is **False** or **UNKNOWN** if its argument is unevaluated.

```
module FWIF where

infixr `seq`

import Types (Tree,Bool)

import Primitives (eq,boolval)  -- eq :: a -> a -> Bool
                                -- boolval :: Bool -> BOOL

fwifdb :: Bool -> Tree -> Tree -> Tree
fwifdb p x y =
    case boolval p of
         TRUE  -> x
         FALSE -> y
         UNKNOWN | x `eq` y  -> x
                 | otherwise -> x `seq` y `seq`
if x `eq` y then x
                                else fwifeval x y

fwifeval (Node1 l1 k1 r1) (Node1 l2 k2 r2) =
    if l1 `eq` l2 then    r' `par` newnode
    else                  l' `par` newnode

    where  l' =      fwifdb p l1 l2
           r' =      fwifdb p r1 r2
           newnode = Node1 l' (if k1 = k2 then k1 else if p then k1 else k2) r'

------------------   ... Code for Node2 and Tip cases ...   -----------------

fwifeval _ _ = if p then x else y
```