Fast Parallel Implementation of Lazy Languages – The EQUALS Experience[1]

Owen Kaser
Department of Mathematics, Statistics and Computer Science
University of New Brunswick at Saint John
Saint John, NB E2L 4L5, Canada.
`owen@unbsj.ca`

Shaunak Pawagi
C.R. Ramakrishnan
I.V. Ramakrishnan
Department of Computer Science
SUNY at Stony Brook, NY 11794.
{`shaunak, cram, ram`}`@cs.sunysb.edu`

R.C. Sekar
Bellcore, 445 South Street
Morristown, NJ 07962.
`sekar@thumper.bellcore.com`

Abstract

This paper describes EQUALS, a fast parallel implementation of a lazy functional language on a commercially available shared-memory parallel machine, the Sequent Symmetry. In contrast to previous implementations, we detect parallelism automatically using strictness analysis. Another important difference between EQUALS and previous implementations is the use of reference counting for memory management, instead of garbage collection. Our implementation shows that reference counting leads to very good scalability, low memory requirements and improved locality. We compare our results with other sequential (SML/NJ) and parallel ($\langle \nu, G\rangle$-machine and GAML) implementations of functional languages.

1 Introduction

It is well known that functional languages offer a conceptually simple vehicle for programming parallel computers. The main reason for this is that expressions may be evaluated in any order, due to the absence of side-effects. Therefore, detection as well as exploitation of parallelism is much simpler than in imperative languages. This fact has been exploited in

[1] A preliminary version of this paper appeared in LFP'92.

many previous parallel implementations such as ALICE [Dar81], FLAGSHIP [WW87], GRIP [PJ87], Buckwheat [Gol88a], the $\langle \nu, G \rangle$-machine [Aug89] and GAML [Mar91]. Whereas ALICE, FLAGSHIP and GRIP make use of specialized hardware, the other three implementations are based on commercially available shared-memory multiprocessors. In this paper, we focus on the latter approach and describe EQUALS, a fast parallel implementation of a lazy functional language[2] on Sequent Symmetry.

One of the earliest lazy parallel implementations on a shared-memory multiprocessor was Buckwheat. It demonstrated the feasibility of parallel implementation, but was not tuned for performance. On the other hand, the $\langle \nu, G \rangle$-machine and GAML showed performance improvement over sequential implementations such as LML [Aug84], starting from two processors. Both these implementations were able to reduce parallel overheads, and consequently their performance continued to improve even when the number of processors was increased to ten or more. However, these implementations do not satisfactorily exploit one of the primary advantages of functional languages for parallel evaluation, namely, automatic detection of parallelism. For instance, the $\langle \nu, G \rangle$-machine and GAML use program annotations as the only means to identify parallelism. These annotations can be quite cumbersome for the programmer. The approach used in Buckwheat relies on strictness information to identify parallelism, but the strictness information used is based on head-normal form (HNF)[3] and is not sufficient (as we show later) to identify significant parallelism in most programs. To alleviate this problem they make assumptions such as *cons* and *append* being strict, which runs counter to the goals of lazy evaluation.

A second problem with the $\langle \nu, G \rangle$-machine and GAML is that they use memory management techniques that do not scale well. The $\langle \nu, G \rangle$-machine uses a sequential garbage collector and its performance figures given in [Aug89] do not include garbage collection times. GAML uses a parallelized garbage collector, but it scales poorly, e.g., one of the garbage collector speedup curves flattens at a speedup of 2 when the number of processors is increased to 7 or 8. The EQUALS implementation overcomes both these drawbacks as follows:

- Automatic detection of parallelism by propagating exhaustive demand as far as possible. We accomplish this using *ee*-strictness analysis developed in [SPR90][4]. As we show in section 2, exhaustive evaluation also increases task granularity by packing several HNF tasks into one task. It also improves sequential performance by avoiding repeated traversals of the graph.
- Using reference counting for memory management. Our implementation results show that it scales well and also has low memory requirements without compromising efficiency.

A brief overview of the EQUALS system is presented below.

[2] In a lazy functional language, only those computations that are necessary to obtain the normal form of an input expression are performed [HL93].

[3] We use HNF and weak HNF interchangeably.

[4] Most of the other strictness methods do not deal with (exhaustive) normal form demands. For the parallelism detection discussed here, any strictness method that deals with normal form demands can be used.

1.1 Overview of the Implementation

The EQUALS system consists of a compiler and a runtime support system. The runtime support system is broadly divided into subsystems for I/O, memory management, and task management. The I/O subsystem reads an input term from the user, parses it, and creates an initial graph structure. After the input term has been evaluated to normal form, the subsystem invokes a pretty-printer to display the result. Since such support facilities are straightforward, they will not be discussed further. Instead, the remainder of the paper describes the compiler, memory management, and task management, beginning with an overview of each of these modules.

The EQUALS compiler uses *ee*-strictness analysis to detect parallelism. This information is used in translating the source program into a combinator-based intermediate language. The intermediate language includes constructs for creating parallel tasks and synchronizing among them. The goal of compiled code is to normalize a given input expression. If multiple subterms need to be evaluated in order to normalize the input expression, these subterms can be evaluated in parallel. Subterms that are evaluated in parallel are taken up by individual tasks, which execute the compiled code on their private stacks. Since there may be many more parallel tasks than the resources available (e.g., processors, shared memory), tasks are created only when they are deemed useful. Some of these decisions are made at compile time itself; for instance, the compiler will never emit code to create a new task, unless it can generate code for work to be done concurrently by the existing task. At runtime, on the other hand, opportunities for additional parallelism will be passed up, if many parallel tasks already exist.

Although these tasks can be executed as UNIX processes, it is very expensive to do so. EQUALS implements a mechanism for managing light-weight tasks, where tasks are executed under the control of evaluator processes (one per processor). All runnable tasks are placed in a global ready queue. If a task needs the value of a subterm that is currently being evaluated by another task, then the first task is suspended, awaiting the evaluation of the subterm. The evaluator then begins execution of a task from the ready queue. Once the evaluation of a term is completed, all tasks awaiting its evaluation are put back in the global queue. The size of the global ready queue is used to share and balance the system load. Task management is discussed in section 5.

Heap space needed for evaluation of a task is allocated out of a block of free space maintained by the corresponding evaluator. When an evaluator runs out of free space, it allocates a block from a global pool. Heap space freed by a task is released into the free space of the corresponding evaluator. When an evaluator accumulates more than a preset amount of free space, it returns the excess to the global pool. Stacks and other structures used by the tasks are also allocated and freed in the same manner. A detailed description of the memory management scheme appears in section 4.

The rest of this paper is organized as follows. The next section elaborates on the issues of parallelism detection and memory management. The EQUALS compiler is described in section 3. Sections 4 and 5 describe our memory and task management schemes. A detailed discussion of performance of EQUALS is presented in section 6. Our results show that sequential performance is comparable to SML/NJ, one of the fastest functional language implementations. The parallel performance of EQUALS is about the same as the $\langle \nu, G \rangle$-machine, even though EQUALS times include memory management whereas the $\langle \nu, G \rangle$-machine times exclude garbage collection time. Results also show that reference counting mechanism scales well, uses less memory and has better memory locality than copying garbage collection. Concluding remarks about our experience with EQUALS appear in section 7.

2 Issues in Parallelism

The novelty the EQUALS implementation among other shared memory implementations lies in the detection and control of parallelism and in memory management. In this section, we elaborate on these issues.

2.1 Detection of Parallelism

There have been two approaches to identifying parallelism in lazy languages. One approach, used in the $\langle \nu, G \rangle$-machine and GAML, requires programs to be annotated for parallelism. These annotations are different from strictness annotations and can be cumbersome since they are always required. Moreover, to ensure that laziness is not compromised, the task scheduler must have a mechanism to ensure that the normal order branch of computation makes progress. This requires preemption of all resources – processor, heap and stacks. An alternative approach is followed in EQUALS and uses strictness information to identify parallel components. Since strictness identifies only those computations that are needed for the input expression to be normalized, no additional mechanism is necessay to ensure progress of normal order branches. This approach has been used in earlier implementations such as [Geo89] and [Gol88b]. However, they could not extract much parallelism from strictness information alone, since their model of computation was based on (repeated) head evaluation and not on exhaustive evaluation. Hence, the strictness information they use also deals with head-normal forms alone[5]. This strictness information is not sufficient (as shown below) to detect significant parallelism in many programs. In order to get sufficient parallelism, they assume that even non-strict functions such as *cons* and *append* are strict.

To illustrate why strictness based upon HNF alone is not sufficient to identify parallelism, consider the *QuickSort* example shown in figure 1. In that example, the function *split* partitions a list (first argument) based on a pivot (second argument) into two lists. The function $qs1$ takes these partitioned lists, sorts the individual lists and puts them together using *append*. Thus $qs(l)$ first splits the list into l_1 and l_2 and subsequently calls $append(qs(l_1), qs(l_2))$. Hence, a HNF demand on qs results in a HNF demand on *append*. By the definition of *append*, a HNF demand on its output results in a HNF demand on its first argument and *no* demand on its second argument. Hence, $qs(l_2)$ would be invoked only after $qs(l_1)$ is completely evaluated. All the parallelism in *QuickSort* arises from sorting both

[5] i.e., it provides information about which arguments of a function are to be head-normalized in order to head-normalize the function application.

$$
\begin{array}{rcl}
qs(x:xs) & \rightarrow & qs1(split(xs,x,nil,nil)) \\
qs(nil) & \rightarrow & nil \\
split(x:xs,y,u,v) & \rightarrow & if(x>y, split(xs,y,x:u,v), \\
 & & \quad split(xs,y,u,x:v)) \\
split(nil,y,u,v) & \rightarrow & <u,y,v> \\
qs1(<x,y,z>) & \rightarrow & append(qs(x), y:qs(z)) \\
append(x:xs,y) & \rightarrow & x:append(xs,y) \\
append(nil,y) & \rightarrow & y
\end{array}
$$

Figure 1: *QuickSort* program (':' denotes the *cons* operator)

the partitioned lists in parallel, and propagating HNF demand alone is unable to extract any of this parallelism.

We now illustrate how even the extreme measure of declaring *append* as strict in both arguments (under HNF demand) does not lead to any significant parallelism. This is because not only is the HNF strictness insufficient to extract much parallelism, but the HNF evaluation mechanism is unable to exploit the parallelism. If *append* is strict in both arguments then $qs(l_1)$ and $qs(l_2)$ could be invoked in parallel. However, $qs(l_2)$ would be evaluated in parallel with $qs(l_1)$ only until their HNFs are obtained. Evaluation of $qs(l_2)$ would then be suspended until *append* consumes all of its first argument – i.e., until $qs(l_1)$ is completely evaluated. Hence little parallelism results even when *append* is declared to be strict. To exploit all the parallelism in *quicksort* while performing repeated HNF evaluations alone, *cons* has to be considered strict – which runs counter to the goals of lazy evaluation.

2.1.1 Propagating NF Demand and its Merits

In EQUALS, we follow the model of exhaustive (NF) evaluation, instead of repeated HNF evaluation. We identify two extents to which a term may be evaluated – to HNF or to NF – based on the context of evaluation (the demand). Observe that if the output of *append* (or *cons*) is demanded in NF then both its arguments are needed in NF. In other words, *append* and *cons* are *ee*-strict (see [SPR90] for details) in their arguments. By propagating NF demand in this manner and utilizing *ee*-strictness information, we can identify all the parallelism in the examples discussed in this paper. The advantages on NF demand propagation are described below.

In previous implementations, tasks compute weak head-normal forms of terms. (Henceforth, we use "terms", "graphs" and "expressions" interchangeably.) However, HNF tasks are typically fine grained and therefore can easily lead to significant overheads. Although this problem can be alleviated to a large extent by a careful design of task management (as is done in [Aug89] and [Mar91]), nevertheless it is advantageous to use larger grained tasks. Use of NF demand (also called exhaustive or *e*-demand) helps achieve this, since it packs several HNF tasks into a single task.

Propagating exhaustive demand also increases the efficiency of sequential evaluation since it avoids repeated closure construction and context switching. For instance, observe that in the QuickSort example, $qs(l)$ eventually reduces[6] to

$$append(append(\cdots append(t_1, t_2)\cdots)$$

If we do propagate only HNF demand, then the request to head normalize the outermost *append* results in another call to head-normalize the inner *append*. This proceeds all the way to the innermost *append*, which then outputs a single element. This element is consumed by the next outer *append* and so on until the top-level *append* outputs one element. The rest of the computation is represented in a closure, which is invoked only after the first element is consumed. In contrast, if we propagate NF demand then the top-level *append* will force complete evaluation of inner *append*, which in turn will force full evaluation of its inner *append* and so on. Hence, we avoid repeated closure constructions and context switches. Moreover, due to NF demand propagation, EQUALS code is similar to that generated by a strict language and hence its sequential performance is comparable to strict implementations. In summary, propagating NF demand leads to:

- easier detection of parallelism
- larger task granularity
- avoidance of repeated closure building and context switching.

We remark that propagation of exhaustive demand does not translate into inability to deal with lazy streams. In particular, functions that induce such behavior do not propagate exhaustive demand, and hence will be evaluated lazily in our system. For instance, consider the term $first10(qs(l))$, where $first10$ returns the first ten elements of its argument list. Since $first10$ does not propagate normal form demand, $qs(l)$ will be evaluated lazily, so that we do not spend time to sort the entire list, but only to identify as many elements as necessary. Even in the context of a function that propagates exhaustive demand, we may like to "force" stream behavior, so that the function produces its output incrementally. In a parallel implementation such as ours, this can be accomplished without sacrificing the advantages of exhaustive demand propagation, by using *vertical parallelism* (i.e., evaluation of a function and its argument in parallel).

2.2 Memory Management

Most previous implementations of lazy languages on shared-memory machines use variants of mark-and-sweep or copying garbage collectors to reclaim storage. Memory management using garbage collection has the following advantages: it is transparent, can manage memory in presence of imperative updates, and furthermore handles variable size allocations while avoiding loss due to fragmentation. However, for parallel evaluation, this approach suffers from several drawbacks. First, the garbage collector scales poorly when the number of processors is increased. This is because there are certain inherently sequential components and hot spots in the copying phase of the collector such as the need to lock *every* structure

[6] This term may not be constructed explicitly in its entirety; parts of it (e.g., its spine) may be on the stack.

before moving[7]. This problem is compounded by the fact that the garbage collectors traverse much of the heap space and consequently produce a considerable amount of paging activity.

Another problem with the garbage collection approach is that it can lead to poor locality of reference, which is important in a virtual memory/cache environment. When evaluating functional programs, we often build structures that are used just once. With garbage collection, this space is not reused until after the next collection. This means that a page may be brought in from the disk, accessed very few times and then written back.

An alternative to mark-and-sweep or copying garbage collectors reclaims memory through the use of reference counting, and is used in EQUALS. Reference counts have been used in other lazy functional language implementations, such as ALFALFA [Gol88b]. However, its efficiency compared to garbage collection and its effectiveness in a parallel implementation have not been established. The EQUALS implementation shows that reference counting avoids memory contention and improves locality due to immediate reclamation and reuse of free space. It also reduces memory use and is very efficient. For instance, our sequential run times are comparable to those of SML/NJ (with dereferencing typically taking less than 20% of the time) and heap space usage is typically 15 to 25% of that used by SML/NJ.

Reference counting implementations are usually limited to acyclic structures only. Note that this is a limitation of individual implementations and not the approach itself. Intuitively, reference counts can be used in presence of cycles by detecting when back edges are created (thus creating cycles) and not counting the back edges. For instance, the general algorithm in [Hug83] collects cyclic structures using reference counts. Unfortunately, the overhead of such algorithms is high since during the addition of every edge, we must check whether this is a back edge. In a lazy, purely functional language such as EQUALS, cyclic data structures can always be coded as output of infinite functions. For example $ones() = 1 : ones(); \;\; x = ones()$ evaluates the same structure as $x = 1 : x$. However, the corresponding cyclic representation is more efficient than the infinite function form (see page 188 in Bird and Wadler's text [BW88]). Note that in declarations of the form $let \;\; x = 1 : x$, creation of back edges can be detected statically, at compile time. Hence we can avoid placing the usual reference increment instruction at such points, thereby naturally handling cycles with no additional overhead.

3 Compiler

An EQUALS program consists of a set of functions defined by pattern match. The abstract syntax of the source language of EQUALS is given in figure 2. Each function definition in the program is translated into a corresponding function in the intermediate language, and its body is translated into a sequence of intermediate code statements. The constructs in the intermediate language are given in figure 3. Note that this language bears some similarities

[7] Although some recent concurrent collectors (e.g., [WGH92]) do not lock when copying, they require additional memory and add overhead to node access and allocation. Moreover, note that collectors such as the Appel-Ellis-Li [AEL88] collector are sequential; the concurrency arises from a single process performing collection while other (mutator) process(es) are executing the program.

$$
\begin{array}{rcl}
\textbf{program} & ::= & \textbf{fundef}; \cdots; \textbf{fundef} \\
\textbf{fundef} & ::= & f(\textbf{pat}, \ldots, \textbf{pat}) = \textbf{expr} \\
\textbf{expr} & ::= & \text{if } \textbf{expr} \text{ then } \textbf{expr} \text{ else } \textbf{expr} \\
 & & \mid d(\textbf{expr}, \ldots, \textbf{expr}) \\
 & & \mid x \\
\textbf{pat} & ::= & c(\textbf{pat}, \ldots, \textbf{pat}) \\
 & & \mid x
\end{array}
$$

Notes:

f: function symbol,

c: constructor,

d: functor or a constructor, and

x: a variable.

Figure 2: EQUALS source language syntax

to the G-machine, but differs in many ways, such as explicitly named variables and functions, and constructs for demand propagation. The target language (C) influenced some constructs; for instance, compound expressions were permitted since they are allowed in C and hence can be more efficiently compiled than an equivalent sequence of simple expressions. Furthermore, the structure of the intermediate language permits compilation to be a simple translation followed by a series of optimization steps.

3.1 Compilation Algorithm

First, the pattern matching constructs of EQUALS are transformed to case expressions, using the Huet-Levy [HL93] algorithm for lazy pattern matching[8]. After pattern matching, the only change to the structure of the source is the introduction of case expressions. The code generator (see figure 4) takes these transformed function definitions and produces intermediate code. It consists of several functions listed below. Most of them take as parameters the fragment of the source program to be translated and *extent*, which specifies the demand on this fragment. The value of *extent* can be UNK, which means that the demand is not known statically, or one of NF or HNF. This parameter *extent* is *not* to be confused with *context*: the former is a *compiler* parameter used to propagate demand *statically* at compile time whereas the latter is a parameter to functions in intermediate code and is used to propagate demand at *run-time*. In the figure, $\mathsf{best}(\mathit{extent}, \mathit{context})$ stands for 'NF' if $\mathit{extent} = NF$ and '*context*' otherwise. The function *GetFreshVariable* generates unique variable names.

$\mathcal{F}$: the top-level code generator for a function.

$\mathcal{E}$: translates an expression. It takes three parameters: the expression to be translated, the name of the variable into which the value of the expression is to be stored (y) and *extent*.

[8] Our current system does not handle prioritized patterns. It can be done using the techniques of Laville [Lav88], Puel and Suarez [PS90] and those in Sekar et al. [SRR92].

Function $f(context, x_1, \ldots x_n)$:
Header for function f. Every function in the intermediate code takes a parameter named *context*. This parameter specifies (at runtime) the extent to which the output of (the current invocation of) a function needs to be evaluated. Using this parameter we propagate demand at runtime.

Assign $var, expr$.

If $expr$ **then** $block_1$ **else** $block_2$.

Switch x { **case** $c_1 : block_1, \ldots,$ **case** $c_n : block_n$ }.

Eval x **to** *context* **at** *location*.

FunctionEval $f(v_1, \ldots, v_n)$ **to** *context* **at** *location* **result** x.

BuildTerm $d(v_1, \ldots, v_n)$ **result** x.

GetChild i **of** x **result** y.

WaitFor *context* **of** x.

Deref x.

Return x.

Notes:

extent : the extent of evaluation (NF or HNF)

expr : constants, variables or compound expressions using predefined functions, such as +

block : a sequence of statements in intermediate code.

location : either *local* or *remote* and specifies whether a task is to be evaluated locally or at a remote site (i.e., on another processor).

Figure 3: EQUALS intermediate language

$\mathcal{P}$: is like $\mathcal{E}$, but handles pattern-matching.

$\mathcal{A}$: generates code for evaluating argument expressions to a function. $\mathcal{A}$ takes five arguments: the expression e to be evaluated, the argument position of e, the name d of the function that has e as an argument, the name of the variable y into which the result is stored and *extent*. $\mathcal{A}$ differs from $\mathcal{E}$ in that it takes the strictness of d into account to determine the demand on e.

$\mathcal{B}$: generates code to build the graph representing its argument e. The result is stored into the argument y.

In order to preserve simplicity, the code generator itself attempts no optimizations. Note that all function evaluations have been labeled for possible remote evaluation, and no synchronization barriers have been placed. Moreover, code for managing the free space (e.g., dereferences) has not been generated. These are generated after a flow analysis is performed during optimization phases. Furthermore, sharing of common sub-expressions, elimination of temporary variables, and unboxing of structures are handled in the subsequent optimization

$$
\begin{array}{lcl}
\mathcal{F}\ \llbracket f(x_1,\ldots,x_n)=e \rrbracket & = & y \leftarrow \mathit{GetFreshVariable}() \\
 & = & \textbf{Function } f(\mathit{context}, x_1,\ldots,x_n); \\
 & & \mathcal{E}\ \llbracket e \rrbracket\ y\ \textbf{UNK}; \\
 & & \textbf{Return } y \\
\mathcal{E}\ \llbracket \textbf{case } x \textbf{ in } (pm_1,\ldots,pm_n) \rrbracket & = & \textbf{Eval } x \textbf{ to HNF at } \mathsf{Remote}; \\
\qquad y\ \mathit{extent} & & \textbf{Switch } x\quad \mathcal{P}\ \llbracket pm_1 \rrbracket\ x\ y\ \mathit{extent}; \\
 & & \qquad\qquad \vdots \\
 & & \qquad\qquad \mathcal{P}\ \llbracket pm_n \rrbracket\ x\ y\ \mathit{extent} \\
\mathcal{E}\ \llbracket x \rrbracket\ y\ \mathit{extent} & = & \textbf{Eval } x \textbf{ to } \mathsf{best}(\mathit{extent}, \mathit{context}) \textbf{ at } \mathsf{Remote}; \\
 & & \textbf{Assign } y, x; \\
\mathcal{E}\ \llbracket \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rrbracket & = & \mathcal{E}\ \llbracket e_1 \rrbracket\ z\ \textbf{NF}; \\
\qquad y\ \mathit{extent} & & \textbf{If } z \textbf{ then } \mathcal{E}\ \llbracket e_2 \rrbracket\ y\ \mathit{extent} \textbf{ else } \mathcal{E}\ \llbracket e_3 \rrbracket\ y\ \mathit{extent} \\
\mathcal{E}\ \llbracket d(e_1,\ldots,e_n) \rrbracket\ y\ \mathit{extent} & = & z_1 \leftarrow \mathit{GetFreshVariable}() \\
 & & \vdots \\
 & & z_n \leftarrow \mathit{GetFreshVariable}() \\
 & & \mathcal{A}\ \llbracket e_1 \rrbracket\ 1\ d\ z_1\ \mathit{extent}; \\
 & & \vdots \\
 & & \mathcal{A}\ \llbracket e_n \rrbracket\ n\ d\ z_n\ \mathit{extent}; \\
 & & \left\{ \begin{array}{ll}
 \textbf{BuildTerm } d(z_1,\ldots,z_n) \textbf{ result } y & \text{if } d \text{ is a } \textit{constructor} \\
 \textbf{Assign } y,\ d(z_1,\ldots,z_n) & \text{if } d \text{ is a } \textit{predefined function} \\
 \textbf{FunctionEval } d(z_1,\ldots,z_n) & \text{if } d \text{ is a } \textit{user-defined function} \\
 \quad \textbf{to } \mathit{best}(\mathit{extent}, \mathit{context}) & \\
 \quad \textbf{at } \mathsf{Remote} \textbf{ result } y &
 \end{array} \right. \\
\mathcal{P}\ \llbracket c(x_1,\ldots,x_n) \rightarrow e \rrbracket\ x\ y\ \mathit{extent} & = & \textbf{case } c: \quad \textbf{GetChild 1 of } x \textbf{ result } x_1; \\
 & & \qquad\qquad \vdots \\
 & & \qquad\qquad \textbf{GetChild n of } x \textbf{ result } x_n; \\
 & & \qquad\qquad \mathcal{E}\ \llbracket e \rrbracket\ y\ \mathit{extent} \\
\mathcal{A}\ \llbracket e \rrbracket\ i\ d\ y\ \mathit{extent} & = & \left\{ \begin{array}{ll}
 \mathcal{E}\ \llbracket e \rrbracket\ y\ \textbf{NF} & \text{if } i^{th} \text{ arg. of } d \text{ is } \textit{ee}\text{-strict and } \mathit{extent} = \textbf{NF} \\
 \textbf{If } \mathit{context} = \textbf{NF} & \text{if } i^{th} \text{ arg. of } d \text{ is } \textit{ee}\text{-strict and } \mathit{extent} \neq \textbf{NF} \\
 \quad \textbf{then } \mathcal{E}\ \llbracket e \rrbracket\ y\ \textbf{NF} & \\
 \quad \textbf{else } \mathcal{B}\ \llbracket e \rrbracket\ y & \\
 \mathcal{B}\ \llbracket e \rrbracket\ y & \text{otherwise}
 \end{array} \right. \\
\mathcal{B}\ \llbracket x \rrbracket\ y & = & \textbf{Assign } y, x \\
\mathcal{B}\ \llbracket d(e_1,\ldots,e_n) \rrbracket\ y & = & z_1 \leftarrow \mathit{GetFreshVariable}() \\
 & & \vdots \\
 & & z_n \leftarrow \mathit{GetFreshVariable}() \\
 & & \mathcal{B}\ \llbracket e_1 \rrbracket\ z_1; \\
 & & \vdots \\
 & & \mathcal{B}\ \llbracket e_n \rrbracket\ z_n; \\
 & & \textbf{BuildTerm } d(z_1,\ldots,z_n) \textbf{ result } y;
\end{array}
$$

Figure 4: **Compilation Scheme**

phases. The optimizations are a collection of known techniques that have been combined to achieve good efficiency.

3.2 Optimizations

1. **Unboxing**: Any boxing operation on a value followed by an unboxing operation is removed, eliminating all unnecessary boxing operations. When a function needs a parameter in a fully evaluated state (which is determined from strictness), it is passed as an unboxed value. Also, functions return unboxed values whenever possible. In the implementation of EQUALS, we found that such inter-procedural unboxing improves speed by more than a factor of 2.

2. **Lifetime Analysis**: A variable is determined to be evaluated to different extents, unevaluated, or in an unknown state at each point in the code, by a lifetime analysis which is performed across basic blocks. Using this analysis, the following optimizations are performed:

   - *Placement of Synchronization Barriers* just before the point where the value is needed.
   - *Remote vs. Local evaluation*: If no significant work is done between a remote evaluation and its corresponding **WaitFor** (e.g., only straight-line code), it is changed to a local evaluation.
   - *Common Subexpression Sharing* is done to share expressions with different evaluation extents and across basic blocks.
   - *Immediate Reclamation of Free Space*: The term pointed to by a variable is dereferenced immediately at the end of that variable's lifetime. Our experience shows that this significantly reduces heap space, e.g. heap-space requirement is brought down by a factor of 4 in *QuickSort*.
   - *Reducing the Number of Temporary Variables* using a graph-coloring heuristic.

3. **Tail Recursion Elimination**: Direct tail recursion and certain linear recursions (with associative functions as outermost symbols) are converted into loops.

4. **Generating two versions**: Finally, we can eliminate the *context* parameter (together with all the tests on its value) by generating two versions of the code for each function. The two versions are invoked when the result is required in NF and HNF respectively.

An example EQUALS program (*nfib*) and its optimized, NF demand version of the intermediate code are given in figure 5. The unoptimized code for *nfib* is given in the appendix, to indicate the effect of the optimizations.

4 Memory Management

Memory is divided into two sections, namely, heap space and stack space. We separate the stacks from the heap, since stacks show greater locality and are simpler to manage. In the following we first describe the implementation of the heap, including node design, allocation policy, and deallocation via reference counting, followed by a description of stack-space management.

```
nfib(n)  = if (n < 2)  then 1
                        else nfib(n-1) + nfib(n-2) + 1
```

Function *nfib_NF*($x1$)
If $x1 < 2$ **then**
    Assign $y1, 1$
else
    Assign $y1, (x1 - 1)$
    FunctionEval *nfib_NF*($y1$) **at** *Remote* **result** $y2$
    Assign $y1, (x1 - 2)$
    FunctionEval *nfib_NF*($y1$) **at** *Local* **result** $y3$
    WaitFor *NF* **of** $y2$
    Assign $y1, (y2 + y3 + 1)$
Return $y1$

Figure 5: An example EQUALS program (top) and its intermediate code (bottom)

Status	Ref. count
Value	
WaitQ Ptr	Ptr to Child 1
Ptr to Child 2	Ptr to Child 3+

Status fields include:

NF: Indicates whether the term rooted at this node is in normal form.

InProcess: Set if the term rooted at this node is in the process of being (head) normalized.

Lock: Used to serialize accesses to the node.

Overflow: Set if the current node has more than three children.

Type: Indicates whether the value of the node is a functor, constructor, integer, float, etc.

Figure 6: Structure of Heap Nodes

4.1 Node Design and Locking

When a term is normalized, it is overwritten with its normal form. If all graph nodes are of equal size, this is easily achieved by overwriting the term's root node. Under this scheme, nodes of arbitrary arity are accommodated using a chain of *overflow pointers* to reference additional children. Some systems, such as the $\langle \nu, G \rangle$-machine, use variable-size nodes; properly overwriting a smaller node with a larger node then requires overwriting with an *indirection node* that points to the larger node. Thus, at runtime, certain node accesses must be checked for indirection. The fixed-size scheme avoids this indirection cost and also enables simple reference-counting collection without fragmentation.

The structure of the graph nodes is given in figure 4.1. If the node has more than three children, the *Overflow* flag is set and the first two child pointers point to the first two children; the third child pointer points to a list of overflow nodes, each of which contain pointers to the other children. The *WaitQ* field points to the notification list of tasks, to be awakened when the (head) normalization of the node is complete.

The locks we have implemented are shadow locks, in which we spin on the copy of the lock bit in cache until it is reset and then try to obtain the lock [Seq87]. These locks use the atomic test-and-set (`btsw`) instruction available on the Sequent and generate less bus traffic than naive locks. In general, a node is accessed only after the lock is acquired. To further reduce locking overheads, the design permits accessing a node without locking under certain (common) conditions. Some of the strategies used to avoid locked access to nodes are described below.

If a node has been normalized, the extent of its normalization guarantees that either the entire term (if in NF), or just the root (if in HNF) will not require locking. The implementation ensures that the status flags indicating the extent of normalization can be safely read without locking. For instance, consider the flag that indicates whether a term is in NF. Even when the term is in an inconsistent state (i.e., partially overwritten), the implementation guarantees that this flag is not set. Thus, it is safe to examine the flag and if set, all the data fields in the node can be safely extracted without locking. Since EQUALS is a lazy language, tests for normalization are done often and the above optimization is very important. Another case when locking is not required arises when a node cannot be referenced by another processor, e.g., when the node has just been allocated. Finally, the reference count is manipulated exclusively with atomic increment and decrement instructions and the normal locking convention is not used.

Our implementation seeks to keep heap nodes reasonably small (32 bytes). Since most of the space is occupied by pointers, we used 16-bit pointers to reduce the size. Being a prototype, the limit on the the number of nodes (2^{16}) is acceptable. (Even this represents a 2Mb heap, half the physical memory of the machine on which development was begun.) Apart from the ceiling on the memory use and the overhead of shift and add instructions on every access, the other disadvantages of this scheme include the need to pad node sizes to a power of two, enlarged object-code size, and the inability to allocate nodes with known lifetimes on the stack. In the future implementations, though, we plan to redesign the nodes to accommodate full-length pointers.

4.2 Heap Allocation

In order to avoid contention when allocating memory, the heap is managed as a two-level structure[9]. The lower-level structure is a linked list of free nodes, called a *block* and the higher-level structure is a locked global pool of blocks.

[9] A two-level allocation strategy, involving garbage collection, was also used in GAML.

Initially, a block from the global pool is given to each evaluator, which privately allocates from (and deallocates to) its block. Blocks are permitted to grow to a certain maximum size, after which they become *full.* When this occurs, the evaluator begins a new block. The full block may either be returned to the global pool, or the evaluator may keep it for possible later use. Blocks may also become empty, if an evaluator allocates more than it deallocates. If the evaluator thus exhausts its current block, it will use a full block that it has kept for such possible use; if it has no such block, it will obtain one from the global pool.

In the current implementation, an evaluator keeps two full blocks on hand before it begins returning them to the global pool. This hysteresis (i.e., empty on zero and full on two) avoids thrashing on the global pool. For instance, without hysteresis (i.e., full on one), when an evaluator has exactly one full block it would return the block to the global pool. If it must next perform a node allocation, immediately it would have to obtain a block from the global pool; hence, an alternating sequence of allocations and deallocations would lead to thrashing at the global pool.

4.3 Reference Counting

As mentioned before, reference counting is used to reclaim free space. Since there is no separate phase in which all processes collect free space, opportunities for contention at memory reclamation are minimized. Moreover, reference counts permit the following trick to avoid locking when a node is freed. Observe that a node about to be freed (i.e., a node being dereferenced with reference count = 1) will be referred to only by the current evaluator. Thus, there is no need to lock it before freeing. Since many dereferences satisfy this condition (e.g., 45% of the dereferences in the *Euler* example), this trick is important in practice. In contrast, since the reference information is not available for a garbage collector, this trick cannot be used to avoid locking at copying time.

Using reference counting we can immediately reclaim freed space. This results in greatly reducing the heap space usage. Furthermore, by maintaining the free list as a LIFO, we immediately reuse memory that is freed. Since nodes are created and destroyed very quickly in typical programs, this strategy greatly increases the chances of using the same set of memory locations repeatedly, thus improving locality.

4.4 Stack Management

The stack space is divided into stacks of several different sizes, and initially each task is allocated a small stack. There can be many suspended tasks at any time, many of them waiting without having performed much computation. Allocating a small stack initially reduces the space wasted by such tasks. Stacks are used in the usual manner during execution of the C code to evaluate a term, and thus an activation record is usually allocated adjacent to its parent in memory, unless overflow occurs.

There are two possible mechanisms to handle stack overflows. In the first, when a task's stack overflows, it is extended by linking another stack. Checking for stack overlow is performed at every entry to a function. At every exit from a function, the stack has to be checked for underflow. In case of an underflow, the current stack is unlinked and the return value passed to the old stack.

In the second approach, the stack is expanded on overflow; there is no need to check for underflow. In this approach, overflow triggers the allocation of a larger stack, and the contents of the smaller stack are moved to the new stack, with appropriate adjustments to stored frame pointers. Moving the old stack's contents can be achieved by adjusting the virtual-to-physical address translation tables. This approach incurs some overhead from copying and adjustment, but it has certain advantages relative to linking; for instance, we need not check for underflow. Moreover, it avoids the possibility of repeatedly linking a new stack, using it minimally, underflowing and unlinking it, and then immediately overflowing the old stack again.

In EQUALS, we used the second approach of expanding stacks, to avoid the overhead of underflow checks. Unfortunately, on the Sequent Symmetry, the virtual-to-physical address translation tables are inaccessible to nonpriviledged programs, and the operating system interface to them imposes unacceptable limitations on the allowable memory layout. Hence, in the EQUALS implementation, we have been forced to copy the contents of the smaller stack onto the larger stack. Initially, tasks are given stacks whose size is 16kb. When overflow is imminent (within 1200 bytes of the stack limit[10]), the stack's contents is copied to a 256kb stack. Later overflowing by the task will, each time, double the size of its stack. Although the initial 16kb stacks are pre-allocated, larger stacks are dynamically allocated. Note that choosing a geometrically increasing sequence of sizes bounds the worst-case overhead of the copying technique.

A major disadvantage of the copying process is the burst of bus traffic, including many writes, that it generates. However, stack overflow is rare and does not seem to be a serious bottleneck[11]. We are currently investigating linked stacks (the first approach), seeking ways to reduce the underflow checks and hysterisis effects, and studying the relative efficiency of the two approaches.

5 Task Management

The task management subsystem provides mechanisms for the creation, synchronization, and load-balancing of tasks. Each of these mechanisms is described in detail below.

5.1 Task Creation

Recall that the purpose of a task is to evaluate a term to either NF or HNF. Task creation consists of building the term to be evaluated (if it does not already exist) and allocating a stack on which to begin its evaluation.

[10]This is enough of a buffer to permit most runtime support routines from having to check for stack overflow.

[11]The sizes chosen for stacks are a factor; in almost all examples discussed later, only the small initial stacks (and not many of them) overflow. Relative to the total accesses made to stack locations, the amount from copying is almost negligible.

In contrast with our scheme for task creation, the $\langle \nu, G \rangle$-machine intermingles the stack and graph, by allocating stack space in each function-apply node in the graph. Since the stack and heap accesses show different behavior (e.g., stacks accesses show better locality) we chose to keep the stack and heap spaces distinct. Furthermore, creating stack frames on the heap requires variable-size graph nodes, leading to fragmentation and its associated problems when reference counting is used.

Task Control Blocks: Each task's information is stored in a task control block that includes not only the task's stack, but also other vital information. It records the term under evaluation, the base and limit of the task's stack, and it has a field to store the stack pointer when the task is suspended. Currently, there is a fixed maximum number, S, of tasks. This lets us pre-allocate task control blocks in an fixed-length array, and again permits use of a shorter task pointer (an index) than would otherwise be possible. This is useful, since each graph node currently requires a field indicating the task processing it. The restriction on the number of tasks could easily be avoided, but we have not observed any insurmountable difficulties arising from it. Since the number is small, it sometimes provides a throttle on the creation of tasks, because we avoid task creation unless a small-stack task is available. (At most 20% of the tasks have large stacks.)

In an early version of EQUALS, this throttle on task creation led to the following interesting situation: a program was executed where one task created another, and quickly suspended awaiting it. Then, the new task behaved similarly. Rapidly, all small tasks were allocated, and so the system switched to a sequential, stack-intensive mode of evaluation. Overflowing its stack, a larger stack was given to the task. Then, its original stack was released to the pool of available stacks. Now, unfortunately, the system was briefly able to switch out of sequential mode, create a new task to run on the newly freed stack, and suspend the task running on the large stack. Having no more stacks, evaluation (again on the small stack) returned to sequential mode, and the process repeated until the the maximum number of tasks was exceeded.

To overcome this problem, hysteresis is used in the current implementation, which uses two thresholds S_{seq} and S_{par}. These thresholds control whether the runtime system will permit creation of new, parallel tasks, or whether it will enforce sequential execution. When parallel-task creation is allowed, it is permitted until more than S_{seq} tasks exist, at which point sequential execution is enforced. Sequential execution then remains enforced until the number of tasks becomes less than S_{par}. We have found that good results are obtained by setting S_{seq} and S_{par} to 80% and 50% of the total number of small stacks respectively.

5.2 Task Synchronization

While evaluating a term t, a task T may find that it must evaluate one of its subterms[12], say s. Task T may create a new task, say S, for this subterm s, or T may evaluate s by itself. Synchronization is clearly required in the first case, since T may have to wait for the

[12] This subterm may not be a part of the original term, but instead be created during its evaluation.

completion of S. Even in the second case, synchronization may be required: If s is already being evaluated by another task, T must suspend itself until s is evaluated. In such cases, T executes a **WaitFor** instruction that, in effect, enters it into a wait queue for s. (The actual mechanism is discussed later in this section.) Its evaluator then proceeds to execute the next task from the global ready queue. Note that there is no need to preempt tasks in EQUALS, since we use a conservative approach that never generates work that is not required.

An important point to be noted here is that wait queues are associated with terms, rather than tasks. This is because a task T created to evaluate a term s may also take up many subterms $s_1, ..., s_k$ of s for (local) evaluation. Suppose that another task T' needs to evaluate s_1. In this case, observe that T' need wait only until s_1 is evaluated. However, T will complete only after evaluating all of $s_2, ..., s_k, s$. Thus, parallelism from simultaneous execution of T and T' is lost if T' waits on T instead of s_1. Also, note that T selectively waits for one or more subterms, and is not restricted to await *all* subterms that it has created before it can resume from any one of them. (This restriction is imposed in [HS92], where it necessitates additional measures to avoid deadlock.)

Since a term may be evaluated to different extents (HNF or NF) depending on the demand, evaluation of shared subterms complicates matters. To see this, consider a term t being evaluated to extent ext_1 by a task T_1. Before its evaluation is complete, suppose a task T_2 needs the same term t to be evaluated, this time to extent ext_2. The following scenarios arise:

- $ext_1 = ext_2$: T_2 is added to the wait queue for t and is awakened when t's evaluation is complete.

- $ext_1 =$ NF and $ext_2 =$ HNF: T_2 is added to t's wait queue, awaiting its NF, since the code executed evaluating t to NF does not produce an intermediate result in HNF. This potentially reduces parallelism, since T_2 is blocked longer then strictly necessary. Nevertheless, efficiency is improved; often, T_2 would soon request subterms of t in HNF, duplicating the work already being done to evaluate t to NF.

- $ext_1 =$ HNF and $ext_2 =$ NF: T_2 is added to t's wait queue, but will not be released when t's HNF is computed. We might create a new task to evaluate t to NF, but to prevent T_1 from possibly overwriting the NF we would have to kill it, and all tasks it has spawned. We avoid this difficulty by permitting T_1 to complete, at which time t is taken up for normalization. Though we might release terms awaiting t's HNF at this time, for efficiency all tasks are made to await t's NF.

Finally, we note that EQUALS implements the notification model of task creation, as opposed to the advisory-sparking method used by a number of recent implementations [Aug89, Geo89, Mar91]. In the advisory method, a task is not created until an evaluator is free to run it: a pointer to the graph to be evaluated would be put into a spark pool by T, but no guarantees are made that a task will ever be created. Rather than block for a sparked graph for which no task has been created, T would, with this method, evaluate the graph itself. The main advantage of this technique is that the spark pool need not be locked; however, the ready queue must still be locked.

Implementation of Synchronization Mechanism: The existing implementation does not use an explicit wait queue to provide synchronization. Initially, we were concerned that the following situation might impair performance too much. In this situation, two tasks have been created, and both must be awaited. Using wait queues, two **WaitFor** instructions would be required in series, possibly leading to unnecessary synchronization, where a task is awakened only to re-suspend immediately. Instead, a more flexible scheme was implemented for the runtime system. Our experience shows, though, that this scheme is usually not required, and examination of the code generated reveals that the full power of the scheme is rarely useful. Thus, future implementations of EQUALS will use the simpler explicit wait queue. However, for completeness we next explain the actual mechanism used.

Consider a task T which requires evaluation of subterm s, and creates a task to do this. Then, T will be placed in a notification list for s, along with a pointer to a memory location in T's stack. Such a memory location is called a *wait counter*, and it is decremented when the task for s completes. Though the wait-counter mechanism supported by the runtime system allows a more general scheme, the compiled code currently uses binary values for wait counters: zero indicates s has completed, and one indicates that it has not. Larger values are possible, indicating more tasks are pending, and all of these tasks must complete before wakeup can occur. Unlike the scheme described by [Geo89], T can have several wait counters: a single wait counter would imply that T must await all tasks it has spawned thus far, even those not strictly required until later. In our scheme, whenever synchronization between T and s is required, T executes a **WaitFor** instruction, which examines the status of s. If s has not yet completed, **WaitFor** marks t as inactive. After s is evaluated, its task will decrement the wait counters in its notification list. If a task's counter becomes zero and it was blocked awaiting this particular counter, the task is then moved to the global ready queue. Note that this does not necessarily happen to all tasks whose wait counters have become zero. Such tasks may be awaiting some other subterm, or may still be actively running. Conceptually, such tasks are *not* in the wait queue for s, although they *do* need to be notified of s's evaluation. The major disadvantage of our current synchronization technique is that it complicates the copying approach to stack overflow, and requires one task to modify the contents of another task's stack, thus necessitating additional locking.

5.3 Load Balancing

In EQUALS, new or resumed tasks are placed in the global ready queue from which free evaluators take up tasks[13]. Thus, the global ready queue is the mechanism for load balancing. To reduce contention at the global queue, we create tasks only when the system is lightly loaded. When the number of tasks in the ready queue exceeds some threshold, the evaluators avoid creating tasks and instead perform the intended computation locally. This technique has been used previously [Mar91, Geo89], and details on the selected thresholds are given later.

[13] A task may be taken up by different evaluators during its lifetime. Thus, all memory accessed by a task, including its stack, must be kept in shared memory.

Note that the above technique reduces task creation, and consequently decreases parallel overheads. However, it has been observed in [MKH90] that this may result in too few tasks being created. Evaluators may become idle when parallel tasks could have been executed. However, this has not been a significant problem in our experience. Our experiments indicate that the evaluators are idle for less than 10% of the time. Even this may be due to the presence of inherently sequential components in the computation.

Ready Queue: The global ready queue has been designed so that serialized access does not create a bottleneck. Consider a simple ready queue that is a linked list of tasks. In order to add or remove entries from the queue, the queue needs to be locked. Hence, the lock for the ready queue can become a system bottleneck: Too much bus traffic is generated when all idle evaluators race one another to retrieve a newly enqueued item. To avoid this difficulty, the current version implements the queue in a novel way, which in spirit resembles the dual queue used by [Geo89].

The queue is viewed as a series of *slots*, and each idle evaluator that arrives at the queue is assigned a unique slot (e.g., next 'free' slot). If that slot holds a task, the evaluator runs this task; note that once a slot is assigned, no locking is needed for dequeueing. If the slot is empty, the evaluator busy waits *on this slot* until it is filled. Note that filling other slots does not affect this evaluator's behavior; hence, evaluators do not race one another. When a task is placed in the queue, it is placed in the next empty slot. Note that such a queue can be easily implemented as a circular array of task pointers, and note also that the locks needed to enqueue and to dequeue are independent. Furthermore, the head (next free slot) and tail (next empty slot) can be advanced with atomic locked-increment instructions. The speedup curves shown in section 6 show that this implementation of the ready queue is not a system bottleneck. (An initial implementation of EQUALS used a simple linked list of tasks, which proved to be a bottleneck and lead to the abovementioned improvement.)

Queue Thresholds: As mentioned earlier, new tasks are generated only when the system load is low, as indicated by a global flag. This flag indicates whether the system is in parallel (low load) or sequential (high load) mode. The system switches between the two modes by comparing the size of the ready queue, N, against two thresholds N_{seq} and N_{par}, as follows: When the system is in parallel mode and N becomes larger than N_{seq}, the system enters sequential mode; in sequential mode, when N falls below N_{par}, the system switches to parallel mode. In parallel mode, the runtime system may be requested to create new tasks. It may refuse, as described earlier, if there already are too many tasks. The values of the two thresholds N_{par} and N_{seq} have been determined through experiments, and though there is no ideal setting for all programs, $N_{\mathrm{par}} = 1$ and $N_{\mathrm{seq}} = 20$ yields good overall results. Note that the flag indicating the system's mode need be changed only when a task is enqueued or dequeued. Furthermore, since this flag is advisory, it can be read or written without any locking. Observe that most accesses are to read this flag and are satisfied by the local cache; thus, this flag is not a system bottleneck.

	EQUALS	SML/NJ
Euler	88.0	104
Nqueens	54.8	42.2
MatMult	19.7	14
Sieve	59.0	33
QuickSort	8.6	4

Table 1: Comparison of EQUALS and SML/NJ. (All timings in secs. on a Sun 3/260)

	EQUALS	$\langle \nu, G \rangle$	GAML
MatMult	22.6	NA	NA
QuickSort	9.5	NA	NA
Euler	116.9	128.4	430
Nqueens	64.0	73.9	467
Nfib	32.1	62.1	213

Table 2: Comparison of EQUALS with $\langle \nu, G \rangle$-machine and GAML. (NA: Not Available)

6 Implementation Results and Discussion

In this section we present the results of our implementation based on example programs adapted from [Geo89, Gol88a, Aug89, SPR90]. First we study the sequential performance of EQUALS and show that it is comparable to that of Standard ML of New Jersey (SML/NJ). Following this we compare our speeds and scalability with that of $\langle \nu, G \rangle$-machine and GAML. We then discuss the impact of reference counting on scalability and performance. In particular, we provide experimental evidence to show that memory requirements are significantly less and that locality is improved.

6.1 Sequential Performance of EQUALS

Table 1 compares the performance of EQUALS to SML/NJ (release 0.75)[14]. SML/NJ is a sequential implementation of SML, a strict language, and is among the fastest functional language implementations. In the table, *Euler* computes the Euler totient function from 1 through 1000. In addition to performing substantial amounts of computation, this program also spends a lot of time creating and destroying lists. *MatMult* computes the product of two 100×100 matrices. *Sieve* computes list of primes between 2 and $10,000$. *QuickSort* sorts a list of 5000 integers, and *Nqueens* finds all solutions to the n-queens problem on a 10×10 board.

Observe that speeds of SML/NJ and EQUALS are comparable in *Euler, MatMult* and *Nqueens.* By propagating exhaustive demand and generating two versions, our code is similar

[14] These figures were first reported in [KPR+92].

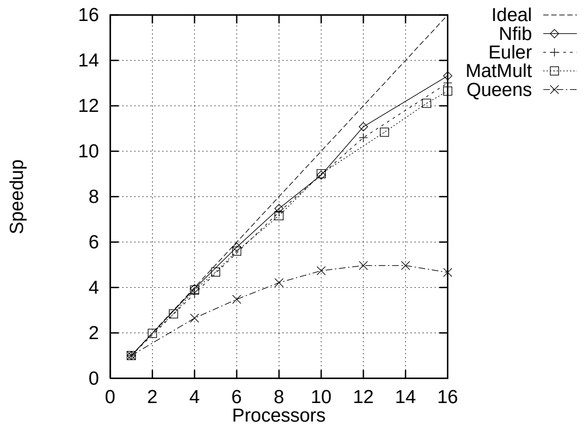

Figure 7: Speedup curves for EQUALS.

to that generated for a strict language, and hence the speeds are comparable. In *QuickSort* and *Sieve*, where there are very few computation steps and most of the time is spent in creating and destroying list structures, SML/NJ is significantly faster because it uses unboxed lists, whereas EQUALS uses boxed lists[15]. This is not a problem in the first three examples, since the number of steps that access lists or perform any other computations are much larger than those that create or destroy lists. (e.g., in *MatMult* there are 10^6 operations of the first kind versus 10^4 list creation/deletion steps.). Boxing can increase the work involved in copying by as much as 100%. Moreover, the performance of EQUALS can be substantially improved by generating assembly code, as is done in SML/NJ. We are quite encouraged to get performance comparable to SML/NJ in spite of these factors.

6.2 Parallel Performance

Table 2 shows wall-clock times for EQUALS, the $\langle \nu, G \rangle$-machine and GAML on a single processor. Timings for both EQUALS and the $\langle \nu, G \rangle$-machine were obtained on Sequent Symmetry with a 16 MHz clock. However, the $\langle \nu, G \rangle$-machine timings do not include garbage collection time, which can account for up to 30% of the total (sequential) time. GAML timings were obtained on a Sequent Balance, which is considerably slower. This impedes a reasonable comparison between our times and those of GAML. However, it is mentioned in [Mar91] that the sequential execution times for GAML are roughly of the same order as those of the $\langle \nu, G \rangle$-machine.

Figure 7 shows speedup curves on all of the examples run using EQUALS. *MatMult* and *Euler* create large grain tasks and hence speedup is almost linear. Although task granularity is very small in *Nfib*, we still scale well, showing that we have managed to keep down task overheads and contention at the global queue. In *Nqueens*, however, we show saturation when number of processors reaches 10, since, in that example, there is a lot of vertical

[15] Currently EQUALS unboxes only primitive data types.

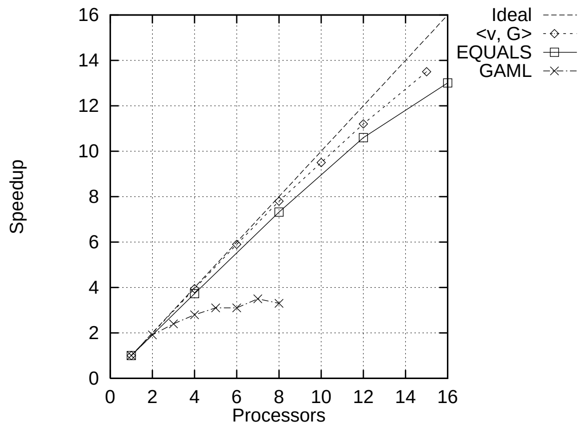

Figure 8: Speedups on Euler.

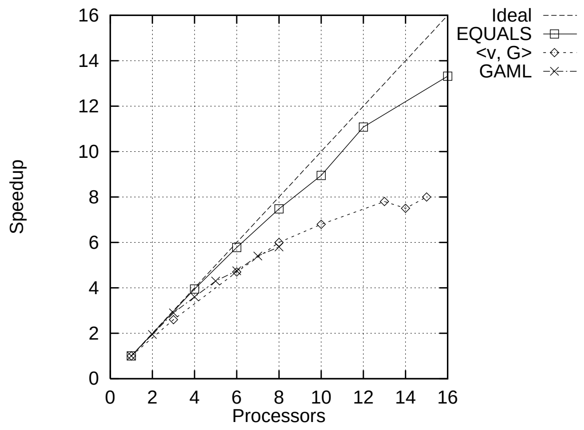

Figure 9: Speedups for Nfib.

parallelism which we do not currently exploit.

Figures 8, 9 and 10 compare the scalability of EQUALS with that of the $\langle \nu, G\rangle$-machine and GAML on *Euler*, *Nfib* and *Nqueens* respectively. Observe that EQUALS scales as well as the $\langle \nu, G\rangle$-machine and GAML on *Nfib*. On *Euler*, it scales as well as the $\langle \nu, G\rangle$-machine and better than GAML. Nevertheless, both $\langle \nu, G\rangle$-machine and GAML scale better in *Nqueens* since they exploit vertical parallelism (unlike EQUALS) with the aid of the advisory-sparking method. Furthermore, the $\langle \nu, G\rangle$-machine timings do not include garbage collection times. As can be seen from the results in GAML, garbage collection times scale poorly, e.g., in *Euler*, the garbage collection time decreases by only a factor of 2 when number of processors increases to 8.

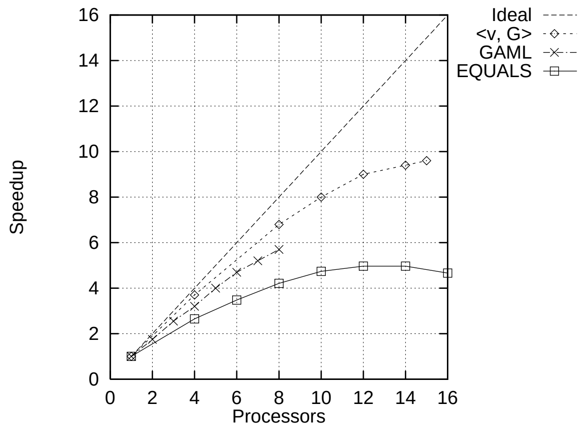

Figure 10: Speedups for Nqueens.

	SML/NJ	EQUALS heap	EQUALS stack
Euler	2.2	0.10	0.07
Nqueens	0.8	0.34	0.09
MatMult	0.8	0.64	0.01
Sieve	2.2	0.32	0.32
QuickSort	1.4	0.16	0.20

Table 3: Memory usage of EQUALS and SML/NJ (in MB's).

6.3 Impact of EQUALS Memory Management

We had mentioned in the introduction that memory management was a crucial component and that by using reference counting we can achieve very good scalability, low memory requirement and improved locality. In this section we give empirical evidence for these claims.

	EQUALS	SML/NJ
Euler	1.66	2.16
Nqueens	2.16	3.12
MatMult	1.77	2.14
Sieve	1.76	2.42
QuickSort	1.97	4.75

Table 4: Ratio of Diskless Sun timings over Server timings.

The *Euler* program spends over 40% of the total time in memory allocation and deallocation, creating and destroying as many as 3 million nodes. The nearly ideal speedup of this program demonstrates the scalability of our memory manager. In contrast, the speedup of GAML appears to saturate even for 5 processors, largely due to poor scaling of the memory management techniques used. Table 3 shows the memory utilization of some programs in EQUALS and SML/NJ. The table shows that EQUALS typically uses substantially less memory than SML/NJ. For EQUALS, we show stack and heap use separately. (Stack usage is less critical, since it is easier to manage and its locality leads to less paging than comparable heap use.) Table 4 shows the relative speeds of SML/NJ and EQUALS programs on a Sun 3/260 (a server), in comparison to a diskless Sun 3/75. The slowdown of EQUALS is near 1.75, which is the factor of difference in raw cpu power between the two machines used. In contrast, the performance of SML/NJ degrades considerably more, due to excessive paging activity. This demonstrates that memory utilization and locality of reference are much better in EQUALS than in SML/NJ. The difference in degradation is large enough to make EQUALS perform better than SML/NJ on most of these examples on a Sun 3/75.

7 Experience with EQUALS

The EQUALS implementation results show it is possible to automatically detect and effectively exploit parallelism in functional programs by propagating exhaustive demand; there is no need to make assumptions such as *cons* and *append* being strict in all contexts. Furthermore, it establishes reference counting as a valid mechanism for memory management. Besides using much less memory and possessing improved locality, reference counting scales well and therefore appears appropriate for parallel implementation.

The implementation experience has also shown us the importance of minimizing task creation and management overheads. We assumed that we can minimize the impact of these overheads by minimizing task creation. We did succeed in reducing task creation: the number of tasks created in EQUALS is less than 10% of the total number that would be created without a throttle on task creation. Still, there is observable parallel overhead, and the task creation time needs to be further reduced.

Moreover, simple and efficient techniques typically perform better than more general and elaborate schemes. For instance, wait counter based synchronization (a generalization of the scheme in [Geo89], where a task waits for multiple subtasks at a single barrier) was initially implemented. Experience showed that most of the waits were performed on a single task and use of the wait counter (with associated overheads of initialization, increment and decrement) was wasteful.

The load balancing scheme used in EQUALS is quite simple, but may not always succeed on more complex programs. We are currently exploring static analysis of programs for sophisticated load balancing. The normal-form demand propagation leads to the problem of efficient exploitation of vertical parallelism, and is a topic of current research. There are several other sources of improvement in EQUALS such as direct generation of assembly code instead of C-code. This will enable us to use registers effectively and reduce the overhead of function calls. We believe that the tighter code can result in considerable performance improvement.

Acnowledgements

This research has been supported by grants from Grumman Data Systems (8476231) and the National Science Foundation (CCR-8805734, CCR-9010269 & CCR-9102159). Use of the Sequent Symmetry S81 was provided by the Department of Computer Science at Rice University under NSF Grant CDA-8619393.

References

[AEL88] A. Appel, J. Ellis and K. Li, *Real-time concurrent collection on stock multiprocessors*, ACM Symp. on Programming Language Design and Implementation, 1988.

[Aug84] L. Augustsson, *A compiler for lazy ML*, Lisp and Functional Programming, 1984.

[Aug89] L. Augustsson and T. Johnsson, *Parallel graph reduction with the $\langle \nu, G \rangle$ machine*, Functional Programming Languages and Computer Architecture, 1989.

[BW88] R. Bird and P. Wadler, *Introduction to Functional Programming*, Prentice Hall, 1988.

[Dar81] J. Darlington, *Alice: A multi-processor reduction engine for the parallel evaluation of applicative languages*, Functional Programming Languages and Computer Architecture, 1981.

[Geo89] L. George, *An abstract machine for parallel graph reduction*, Functional Programming Languages and Computer Architecture, 1989.

[Gol88a] B. Goldberg, *Buckwheat: Graph reduction on shared-memory multiprocessor*, Lisp and Functional Programming, 1988.

[Gol88b] B. Goldberg, *Multiprocessor execution of functional programs*, PhD Thesis, Yale Univ. Dept of Computer Science, YALEU/DCS/RR-618, 1988.

[HL79] G. Huet and J.J. Levy, *Computations in nonambiguous linear term rewriting systems*, Tech. Rep. No. 359, IRIA, Le Chesney, France, 1979.

[HL93] L. Huelsbergen and J. Larus, *A concurrent copying garbage collector for languages that distinguish immutable data*, Principles and Practice of Parallel Programming, 1993.

[Hug83] R.J.M. Hughes, *The design and implementation of programming languages*, Dphil Thesis, Oxford University Computing Laboratory, July 1983.

[HS92] S. Hwang and D. Rushall, *The nu – STG machine: A parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture, draft version*, 4th Workshop on Parallel Implementations of Functional Languages, Aachen, 1992.

[Joh84] T. Johnsson, *Efficient compilation of lazy evaluation*, ACM Symposium on Compiler Construction, 1984.

[KPR+92] O. Kaser, S. Pawagi, C.R. Ramakrishnan, I.V. Ramakrishnan, R.C. Sekar, *Fast Parallel Implementation of Lazy Languages – The* EQUALS *Experience*, Lisp and Functional Programming, 1992.

[Lav88] A. Laville, *Implementation of lazy pattern matching algorithms*, European Symposium on Programming, LNCS 300, 1988.

[Mar91] L. Maranget, *GAML: A parallel implementation of lazy ML*, Functional Programming Languages and Computer Architecture, 1991.

[MKH90] E. Mohr, D. Kranz and R. Halstead, *Lazy task creation: A technique for increasing the granularity of parallel programs*, IEEE Trans. on Parallel and Distributed Systems, 1991.

[PJ87] S. L. Peyton Jones, *GRIP: A parallel graph reduction machine*, Functional Programming Languages and Computer Architecture, 1987.

[PS90] L. Puel and A. Suarez, *Compiling pattern matching by term decomposition*, Lisp and Functional Programming, 1990.

[SPR90] R.C. Sekar, S. Pawagi, I.V. Ramakrishnan, *Small domains spell fast strictness analysis*, ACM Symposium on Principles of Programming Languages, 1990.

[SRR92] R.C. Sekar, R. Ramesh and I.V. Ramakrishnan, *Adaptive pattern matching*, Intl. Conf. on Automata, Languages and Programming, 1992.

[Seq87] Sequent Computer Systems, *Sequent guide to parallel programming*, 1987.

[WW87] P. Watson and I. Watson, *Evaluating functional programs on the FLAGSHIP machine*, Functional Programming Languages and Computer Architecture, 1987.

[WGH92] J. Wild, H. Glaser, and P. Hartel, *Statistics on storage management in a lazy functional language implementation*, Fourth Workshop on Parallel Implementation of Functional Languages, Aachen, 1992.

Appendix

A Compiled code for *nfib*

The code generated using the compilation of rules of figure 4 for the *nfib* program (figure 5) is given below. Note that all **Eval**'s and **FunctionEval**'s are marked *Remote*, no **WaitFor**'s have been generated, and common subexpressions are not eliminated.

```
Function nfib(x1)
Eval x1 to NF at Remote
Assign y1, x1
BuildTerm 2 result y2
If (UnboxInt(y1) < UnboxInt(y2)) then
      BuildTerm 1 result y4
      Assign y3, y4
else
      Eval x1 to NF at Remote
      Assign y5, x1
      BuildTerm 1 result y6
      Assign y7, BoxInt(UnboxInt(y5) − UnboxInt(y6))
      FunctionEval nfib_NF(y7) at Remote result y8
      Eval x1 to NF at Remote
      Assign y9, x1
      BuildTerm 2 result y10
      Assign y11, BoxInt(UnboxInt(y9) − UnboxInt(y10))
      FunctionEval nfib_NF(y11) at Remote result y12
      Assign y13, BoxInt(UnboxInt(y8) + UnboxInt(y12))
      BuildTerm 1 result y14
      Assign y15, BoxInt(UnboxInt(y13) + UnboxInt(y14))
      Assign y3, y15
Assign y16, y3
Return y16
```

Optimizing this code results in the following code, which was shown in figure 5.

```
Function nfib_NF(x1)
If x1 < 2 then
      Assign y1, 1
else
      Assign y1, (x1 − 1)
      FunctionEval nfib_NF(y1) at Remote result y2
      Assign y1, (x1 − 2)
      FunctionEval nfib_NF(y1) at Local result y3
      WaitFor NF of y2
      Assign y1, (y2 + y3 + 1)
Return y1
```