J. Functional Programming **7** (2): 183–217, March 1997. Printed in the United Kingdom

EQUALS – *a fast parallel implementation of a lazy language*

OWEN KASER

Department of Mathematics, Statistics and Computer Science,
University of New Brunswick at Saint John,
Saint John, New Brunswick NB E2L 4L5, Canada

C. R. RAMAKRISHNAN, I. V. RAMAKRISHNAN

Department of Computer Science, SUNY at Stony Brook,
Stony Brook, NY 11794-4400, USA

R. C. SEKAR

Department of Computer Science,
Iowa State University, Ames, IA 50010, USA

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 propagate normal form demand at compile time as well as run time, and detect parallelism automatically using strictness analysis. The EQUALS implementation indicates the effectiveness of NF-demand propagation in identifying significant parallelism and in achieving good sequential as well as parallel performance. Another important difference between EQUALS and previous implementations is the use of reference counting for memory management, instead of mark-and-sweep or copying garbage collection. Implementation results show that reference counting leads to very good scalability and low memory requirements, and offers sequential performance comparable to generational garbage collectors. We compare the performance of EQUALS with that of other parallel implementations (the $\langle v, G\rangle$-machine and GAML) as well as with the performance of SML/NJ, a sequential implementation of a strict language.

Capsule Review

Implicitly parallel functional languages have been a popular area of research for many years. Effective detection of implicit parallelism is a particularly important problem in lazy languages. In this paper, the authors describe EQUALS, a lazy functional system that uses *ee-strictness analysis* to exploit more parallelism than conventional strictness analysis. By detecting when expressions need to be fully reduced, the system can evaluate them to a greater extent than weak- head normal form. Another novel feature is the use of reference counting for memory management.

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 many previous parallel implementations such as ALICE (Darlington and Reeve, 1981), FLAGSHIP (Watson and Watson, 1987), GRIP (Peyton Jones *et al.*, 1987), Buckwheat (Goldberg, 1988a), the $\langle v, G\rangle$-machine (Augustsson and Johnsson, 1989) and GAML (Maranget, 1991). 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.

Buckwheat (Goldberg, 1988a) was among the earliest parallel implementations of lazy functional languages, and demonstrated the feasibility of parallel implementation, but was not tuned for performance. On the other hand, the $\langle v, G\rangle$-machine and GAML showed performance improvement over sequential implementations such as LML (Augustsson, 1984), 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. The $\langle v, G\rangle$-machine and GAML use program annotations as the only means to identify parallelism, and the generation of these annotations was not addressed in these systems. In contrast, Buckwheat automatically identified parallelism using strictness information. However, the strictness information was based only on weak head-normal forms (WHNF), and little parallelism can be identified in many programs using strictness information based only on WHNF (as we show in section 2). To alleviate this problem, assumptions such as *cons* and *append* being strict are typically made, and this clearly runs counter to the goals of lazy evaluation.

Another issue that was left open in both the $\langle v, G\rangle$-machine and GAML implementations concerns the use of scalable memory management techniques. The $\langle v, G\rangle$-machine implementation reported by Augustsson and Johnsson (1989) used a sequential garbage collector. The GAML implementation (Maranget, 1991) used a parallelized garbage collector, but the overhead due to the collector severely limits scalability (e.g. see Maranget, 1991, Figure 8).

These open issues are addressed in the EQUALS implementation as follows:

- *Parallelism is automatically detected and exploited by propagating exhaustive (normal form, or NF) demand as far as possible, at compile time as well as run time.* Parallelism is detected by using *ee*-strictness analysis[1] developed in Sekar *et al.* (1990). Our results show that sequential performance is comparable to

[1] A function is said to be *ee*-strict in an argument if we need to normalize this argument to normalize any application of the function.

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

SML/NJ, one of the fastest functional language implementations. Furthermore, the results indicate that significant parallelism can be detected and exploited automatically, obviating the need for *mandatory* user annotations. The parallel performance of EQUALS is similar to that of the $\langle v, G\rangle$-machine, even though EQUALS times include memory management whereas the $\langle v, G\rangle$-machine times do not.

- *Memory management is based on reference counting*. Our implementation results show that the reference-counting mechanism scales well. Furthermore, this scheme has low memory requirements, good locality and good sequential performance. As a result, the overall performance of EQUALS is comparable to systems that use modern generational garbage collectors.

A brief overview of the EQUALS system is presented below.

1.1 Implementation overview

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, and the information obtained is used in compiling the source program into a combinator-based intermediate language. The goal of this compiled code is to normalize a given input expression, and if multiple subterms must be evaluated to accomplish this goal, these subterms can be evaluated in parallel. Thus, the intermediate language includes constructs for creating parallel tasks and synchronizing among them. Subterms that are evaluated in parallel are taken up by individual tasks, which execute 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 sufficient parallel resources are not available.

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 guide the sharing and balancing of the system load.

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.

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 the memory and task management schemes used in EQUALS and the refinements that were made when the initial system was redesigned to improve performance. A detailed discussion of the performance of EQUALS is presented in section 6. Concluding remarks appear in section 7.

2 Issues in parallelism

The novelty in the EQUALS implementation 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 $\langle v, G\rangle$-machine and GAML, requires programs to be annotated for parallelism. These annotations are different from strictness annotations, and to ensure that laziness is not compromised, the task scheduler typically needs to have a mechanism to ensure that the normal-order branch of computation makes progress. This requires preemption of resources such as the processor, heap and stacks. An alternative approach is to derive parallelism annotations based on strictness, and is used in EQUALS. Since strictness identifies only those computations that are needed for the input expression to be normalized, no additional mechanism is necessary to ensure progress of normal-order branches. This approach has been used in earlier implementations such as those reported by George (1989) and Goldberg (1988b). However, the strictness information used in these early implementations deals only with head-normal forms, i.e. it provides information about which arguments of a function are to be head-normalized in order to head-normalize the function application. This strictness information is not sufficient (as shown below) to detect significant parallelism in many programs. To get sufficient parallelism, they assume that even non-strict functions such as *cons* and *append* are strict.

To illustrate why strictness based on WHNF 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 *qs*1 takes these partitioned lists, sorts the individual lists

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

$$
\begin{array}{rcl}
qs(x : xs) & = & qs1(split(xs, x, nil, nil)) \\
qs(nil) & = & nil \\
split(x : xs, y, u, v) & = & if\ x > y\ then\ split(xs, y, x : u, v) \\
 & & \qquad else\ split(xs, y, u, x : v) \\
split(nil, y, u, v) & = & \langle u, y, v \rangle \\
qs1(\langle x, y, z \rangle) & = & append(qs(x), y : qs(z)) \\
append(x : xs, y) & = & x : append(xs, y) \\
append(nil, y) & = & y
\end{array}
$$

Fig. 1. The program *QuickSort*.

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 WHNF demand on qs results in a WHNF demand on *append*. By the definition of *append*, a WHNF demand on its output results in a WHNF 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 the partitioned lists in parallel, and propagating WHNF demand alone is unable to extract any of this parallelism.

Even declaring *append* as strict in both arguments (under WHNF demand) does not lead to any significant parallelism, since not only is the WHNF strictness insufficient to *extract* much parallelism, but the WHNF 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 WHNFs 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 WHNF evaluations alone, one has to take the extreme measure of annotating *cons* as strict – a step which clearly runs counter to the goals of lazy evaluation.

2.2 Propagating NF demand and its merits

In EQUALS, we propagate normal form (NF) demand and use normal form evaluation as long as such propagation does not affect the termination properties of the program. This contrasts with most other implementations, which are based primarily on weak-head-normal form evaluation. Specifically, in EQUALS we identify two extents to which a term may be evaluated – to WHNF 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 Sekar *et al.* (1990) for details) in their arguments. By propagating NF demand in this manner and utilizing *ee*-strictness information, we are able to identify parallelism in the all examples discussed in this paper.

Another advantage of NF-demand propagation is that it can increase task granularity, leading to lower parallel overheads. In previous implementations, tasks

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

compute weak head-normal forms of terms. (Henceforth, we use 'terms', 'graphs' and 'expressions' interchangeably.) However, WHNF tasks are typically fine grained and therefore can easily lead to significant overheads. This problem can be alleviated to a large extent by a careful design of task management, as done in the implementations of the $\langle v, G \rangle$-machine and GAML. Nevertheless, it is advantageous to use larger grained tasks. Use of NF demand (also called exhaustive or *e*-demand) helps achieve this, since it bundles many WHNF evaluations within 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 to

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

If we propagate only WHNF 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. Consequently, the code generated by EQUALS 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, and
- avoidance of repeated closure building and context switching.

2.3 Run-time propagation of NF-demand and laziness

In this section we consider the interaction between exhaustive-demand propagation and lazy computation. We show that lazy streams, for instance, remain possible. We also show that one can benefit from exhaustive-demand propagation even in the presence of laziness, due to run-time propagation.

Propagation of exhaustive demand does not translate into inability to deal with lazy streams. In particular, functions that induce such behaviour do not propagate exhaustive demand, and hence will be evaluated lazily in our system. For instance, consider the expression

```
take 5 infinite_list_with_complicated_elements
```

where *take* is defined as

```
take (n, x:xs) = if n==0 then nil else x:take(n−1,xs)
```

Clearly, *take* is not *ee*-strict on its second argument, hence the second argument is not evaluated under NF-demand. Thus, laziness is preserved by evaluating the

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

elements of the second argument only on demand, effectively achieving the necessary stream behaviour[2].

From the above example, it may appear that one non-strict function would be sufficient to block the propagation of NF-demand. Although this may be the case when demand propagation is done only at compile time, NF-demand can be safely propagated *at runtime* even beyond non-strict functions. For example, in the above example, propagation of NF demand stopped at the top level. However, after evaluating the *if* statement in the body of *take*, we end up with the term

(complicated_element):take(4, rest_of_the_infinite_list_with_complicated_elements)

The top-level NF-demand can now be propagated on this expression; this results in an NF-demand on each argument of the *cons*, which is *ee*-strict in both arguments. Thus *complicated_element* is not only evaluated with an NF-demand, but it can be evaluated *in parallel* with the rest of *take*, that is, the needed elements of the infinite list. Note that it is the *runtime* propagation of NF-demands that allows selective demand propagation, based on the outcome of conditionals and pattern matches. Thus, demand propagation can be done safely even beyond non-strict functions.

Of course, use of more precise strictness information, such as head and tail strictness, can yield additional benefit, both in terms of additional parallelism and in terms of avoiding construction of closures. However, this added benefit is also associated with a cost – that of evaluation to many different extents and propagation of many different demands at run-time – both of which could complicate the compiler and run-time systems. For this reason, we have deferred the use of more precise strictness information in EQUALS.

2.4 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, can handle variable-size allocations while avoiding the disadvantages of fragmentation. However, this approach seems unsuitable for parallel evaluation, since garbage collectors scale 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 before moving[3]. This

[2] Even in the context of a function that propagates exhaustive-demand, we may wish to 'force' stream behaviour, so that the function produces its output incrementally. In a parallel implementation, 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. However, EQUALS does not exploit vertical parallelism yet.

[3] Although some recent concurrent collectors (e.g. see Huelsbergen and Larus (1993)) 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 collector (Appel *et al.*, 1988) are sequential; the concurrency arises from a single process performing collection while other (mutator) processes are executing the program.

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.

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 (Goldberg, 1988b). However, this technique's 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 efficient. For instance, the sequential run times of EQUALS are comparable to those of SML/NJ (with reference counting typically taking less than 20% of the time) and heap space used by EQUALS is, on the average, only 25% of that used by SML/NJ.

Reference-counting implementations are usually limited to acyclic structures, but this is a limitation of individual implementations and not the approach itself. Intuitively, reference counts can be used in presence of cycles by keeping only one reference count field for all nodes in a strongly connected component (SCC) of a cyclic term; this field stores the number of references to any node in the SCC from some node outside the SCC. The general algorithm by Hughes (1982) collects cyclic structures using reference counts based on this strategy. Unfortunately, the overhead of such algorithms is high, since during the addition of every edge, we must check whether this edge creates an SCC. There is an additional overhead, since at every update, we must identify which reference count field – the node's or the SCC's – is to be updated.

In a 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 preserves sharing, and hence is more efficient than the infinite function form (Bird and Wadler, 1988, p. 188). While our current implementation does not use the cyclic representation, note that in declarations such as $let\ \ x = 1 : x$, creation of SCC's can be detected statically, *at compile time*. Thus, we can use the algorithm due to Hughes (1982) without the overhead of SCC detection. However, since the algorithm still has the additional overhead of identifying the correct reference count field, the performance impact of creating cyclic structures is unclear, and needs to be established through experiments.

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 to the G-machine, but differs in many ways, such as explicitly named variables and functions, and con-

program	::=	[**fundef**]*
fundef	::=	[f(**pat**, ..., **pat**) = **expr**]$^+$
expr	::=	if **expr** then **expr** else **expr**
		| d(**expr**, ..., **expr**)
		| x
pat	::=	c(**pat**, ..., **pat**)
		| x

Notes.
f: function symbol; c: constructor; d: function symbol or a constructor; x: a variable

Fig. 2. EQUALS source language syntax.

Function f(`context`, $x_1, \ldots x_n$)
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** *extent* **at** *location*
FunctionEval $f(v_1, \ldots, v_n)$ **to** *extent* **at** *location* **result** x
BuildTerm $d(v_1, \ldots, v_n)$ **result** x
GetChild i **of** x **result** y
WaitFor *extent* **of** x
Deref x
Return x

Notes.

extent : the extent of evaluation (NF, WHNF or `context`).
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).

Fig. 3. EQUALS intermediate language.

structs for demand propagation. 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, and is used to propagate demand at runtime.

The intermediate code is subsequently translated into C, the final target language. The target language influenced some constructs; for instance, compound expressions were permitted since they are allowed in C and hence may 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 algorithm (Huet and Levy, 1991) for lazy pattern matching[4]. 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 WHNF. The 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, best(*extent*, `context`) stands for 'NF' if *extent* = *NF* and '`context`' otherwise. The function *GetFreshVariable* generates unique variable names.

The compilation scheme uses the translation functions $\mathscr{F}$, $\mathscr{E}$, $\mathscr{P}$, $\mathscr{A}$ and $\mathscr{B}$. An overview of these functions is given below:

$\mathscr{F}$ – generates the code for a function.

$\mathscr{E}$ – translates an expression. It takes three parameters: the expression to be translated, the name of the variable in which the value of the expression is to be stored, and *extent*.

$\mathscr{P}$ – is like $\mathscr{E}$, but handles pattern-matching.

$\mathscr{A}$ – generates code for evaluating arguments passed to a function. $\mathscr{A}$ takes five arguments:

1. the expression e to be evaluated,
2. the argument position of e,
3. the name d of the function that has e as an argument,
4. the name of the variable y into which the result is stored, and
5. *extent*.

$\mathscr{A}$ differs from $\mathscr{E}$ in that it takes the strictness of d into account to determine the demand on e.

$\mathscr{B}$ – generates code to build the graph representing a given expression (first argument); the root of the graph is stored in the given variable (second argument).

To preserve the simplicity of the code generator, many optimizations are deferred until a later stage – the code generator itself concentrates on propagating demand at compile time and generating code to propagate demands at runtime. Furthermore, some instructions that guide the efficient use of resources are not even generated. For instance, a **WaitFor** instruction is needed to coordinate parallel tasks, but such instructions are not initially generated. Instead, we rely on a subsequent optimization

[4] Our current system does not handle prioritized patterns. It can be done using the techniques of Laville (1988), Puel and Suarez (1990) or Sekar *et al.* (1992).

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

$\mathscr{F}$ $[\![f(x_1,\ldots,x_n) = e]\!]$ =
  $y \leftarrow GetFreshVariable()$
  Function $f(\texttt{context}, x_1,\ldots,x_n)$;
  $\mathscr{E}$ $[\![e]\!]$ y **UNK**;
  Return y

$\mathscr{E}$ $[\![$**case** x **in** $(pm_1,...,pm_n)]\!]$ y *extent* =
  Eval x **to WHNF at** Remote;
  Switch x $\mathscr{P}$ $[\![pm_1]\!]$ x y *extent*;
  $\vdots$
  $\mathscr{P}$ $[\![pm_n]\!]$ x y *extent*

$\mathscr{E}$ $[\![x]\!]$ y *extent* =
  Eval x **to** best(*extent*, `context`) **at** Remote;
  Assign y, x;

$\mathscr{E}$ $[\![$**if** e_1 **then** e_2 **else** $e_3]\!]$ y *extent* =
  $\mathscr{E}$ $[\![e_1]\!]$ z **NF**;
  If z **then** $\mathscr{E}$ $[\![e_2]\!]$ y *extent* **else** $\mathscr{E}$ $[\![e_3]\!]$ y *extent*

$\mathscr{E}$ $[\![d(e_1,\ldots,e_n)]\!]$ y *extent* =
  $\{z_1, z_2, \ldots, z_n\} \leftarrow GetFreshVariables()$
  $\mathscr{A}$ $[\![e_1]\!]$ 1 d z_1 *extent*;
  $\vdots$
  $\mathscr{A}$ $[\![e_n]\!]$ n d z_n *extent*;

BuildTerm $d(z_1,\ldots,z_n)$ **result** y	if d is a *constructor*
Assign y, $d(z_1,\ldots,z_n)$	if d is a *predefined function*
FunctionEval $d(z_1,\ldots,z_n)$ **to** best(*extent*, `context`) **at** Remote **result** y	if d is a *user-defined function*

$\mathscr{P}$ $[\![c(x_1,\ldots,x_n) \rightarrow e]\!]$ y *extent* =
  case c : **GetChild** 1 **of** x **result** x_1;
  $\vdots$
  GetChild n **of** x **result** x_n;
  $\mathscr{E}$ $[\![e]\!]$ y *extent*

$\mathscr{A}$ $[\![e]\!]$ i d y *extent* =

$\mathscr{E}$ $[\![e]\!]$ y **NF**	if i^{th} arg. of d is *ee*-strict and *extent* = **NF**
If `context` = **NF** **then** $\mathscr{E}$ $[\![e]\!]$ y **NF** **else** $\mathscr{B}$ $[\![e]\!]$ y	if i^{th} arg. of d is *ee*-strict and *extent* $\neq$ **NF**
$\mathscr{B}$ $[\![e]\!]$ y	otherwise

$\mathscr{B}$ $[\![x]\!]$ y =
  Assign y, x

$\mathscr{B}$ $[\![d(e_1,\ldots,e_n)]\!]$ y =
  $\{z_1, z_2, \ldots, z_n\} \leftarrow GetFreshVariables()$
  $\mathscr{B}$ $[\![e_1]\!]$ z_1;
  $\vdots$
  $\mathscr{B}$ $[\![e_n]\!]$ z_n;
  BuildTerm $d(z_1,\ldots,z_n)$ **result** y;

Fig. 4. Compilation scheme.

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

$$\begin{array}{lll} nfib(n) & = if\ (n < 2) & then\ 1 \\ & & else\ nfib(n-1)+nfib(n-2)+1 \end{array}$$

Fig. 5. An example EQUALS program, *nfib*.

```
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
```

Fig. 6. Compiled code for *nfib* before optimizations.

phase to introduce resource-management instructions, and remove inefficiencies in the generated code.

The code generated using the compilation rules of figure 4 for the program *nfib* in (figure 5) appears in figure 6. Note that all **Eval**'s and **FunctionEval**'s have been marked *Remote*, no **WaitFor**'s have been generated, and common subexpressions have not been eliminated.

3.2 Optimizations

Since the compilation process has been kept simple, the generated code may contain many redundant and inefficient operations. Several optimizations are possible, and indeed, necessary to obtain a practical system. The optimizations form a collection of techniques, both known and novel, that have been combined to considerably improve the performance of the generated code. Some of the most effective optimizations performed are mentioned next, and later explained in more detail.

- *Unboxing* to eliminate most of the unnecessary boxing operations;

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

- *Placing Synchronization Barriers* just before the point where a value is needed;
- *Forcing Local evaluation* whenever no significant work is done between a remote evaluation and its corresponding synchronization barrier;
- *Sharing Common Subexpression* with different evaluation extents and across basic blocks;
- *Reducing the Number of Temporary Variables* to shrink activation records (and hence, improve locality);
- *Reclaiming Free Space Immediately* by deallocating a term immediately after its last access;
- *Eliminating Tail and Linear Recursion* by conversion to loops (see Arsac and Kodratoff (1982)); and,
- *Generating two versions* of code, one for each demand, to reduce repeated tests for demand.

To consider the unboxing optimization further, note that the code generated by the compilation algorithm in figure 4 treats all values as *boxed*, that is, represented by graphs. The efficiency of programs manipulating unboxable values such as integers can be considerably improved by an *unboxing* optimization that removes unnecessary boxing operations. We use a transformational scheme similar to the one presented in Peyton Jones and Launchbury (1991) and remove, for instance, any unboxing operation on a value that is obtained by boxing another value available in this function. This scheme is extended to perform inter-procedural unboxing as follows. When a function receives a parameter of an unboxable type in a fully evaluated state (as determined by strictness), the parameter is passed in unboxed form. In addition, if a function's return type permits, an unboxed value is returned. While implementing EQUALS, we found that such an inter-procedural unboxing optimization improves speed by more than a factor of 2 on examples such as *Nqueens*.

Optimizations involving the placement of synchronization barriers, common subexpression sharing, and reducing the number of temporary variables are all based on the lifetime of variables in the generated code. Using a lifetime analysis, we can determine the extent to which each variable (or more precisely, the term pointed to by the variable) is evaluated, at each point in the code. This lifetime analysis is an intra-procedural flow analysis that is performed across basic blocks; after performing it, placement of synchronization barriers becomes trivial. Also, the optimization of sharing common subexpression evaluations across different evaluation extents is easier with this lifetime information. This optimization uses the observation that the result of a term t evaluated to a demand d can be reused whenever t needs to be evaluated to an equal or weaker demand d'. Finally, the number of temporary variables is reduced by applying a graph coloring heuristic with the knowledge of which variables have overlapping lifetimes. The lifetime analysis itself, as well as the subsequent optimizations are either well known, or are straightforward, and are not described any further.

Creation of unnecessary tasks is reduced by further optimizing the code as follows. Note that it is not worthwhile to create a child task unless the parent has significant work to perform in parallel with the child. Thus, the instructions between

a remote **FunctionEval** or a remote **Eval** statement and its corresponding **WaitFor** are examined. If they do not represent substantial work for the parent, then the **FunctionEval** or **Eval** statement is re-annotated for local evaluation. In place of a sophisticated work-load analysis, the compiler currently assumes that evaluation of a recursive, or unknown, function requires substantial work. In addition, functions which might (in a strict language) call a function requiring substantial work, are themselves deemed to require substantial work. Clearly, a more sophisticated analysis would improve this optimization.

Another optimization significantly reduces the number of live heap cells by deallocating a term as soon as it has been accessed for the last time. This is achieved by *dereferencing*[5] the term pointed to by a variable immediately at the end of that variable's lifetime, even when the variable is a parameter. Thus, a term is deallocated even when a (now dangling) pointer to the term exists on the stack, as long as the pointer is never used again. Compared to approaches where the caller is responsible for dereferencing parameters, this approach significantly reduces memory use. For example, we find that this scheme brings down the heap-space requirement of *QuickSort* by a factor of 4. To reduce reference counting updates, we use a simple technique based on intra-procedural flow analysis. In the future, we plan to investigate the use of one of the analyses that have been proposed in the literature (e.g. see Hudak (1987) or Park and Goldberg (1995)).

Finally, we generate two versions of code for each function, one for each demand (NF or WHNF) on the return value. (Conceptually, we specialize each compiled function on its context parameter, using the two possible values that the parameter may assume.) This optimization eliminates the `context` parameter, and all tests on its value. It should be noted that the NF-demand versions are often open to more optimization, and are usually smaller, than the corresponding WHNF-demand versions. Also, since the numerous tests for context in the single-version code have been eliminated, the space increase due to multiple versions is rather small. Moreover, note that only one version is required for functions that return non-structure values (e.g. integers).

The optimized intermediate code for the function *nfib* given in figure 7 illustrates the effect of the optimizations.

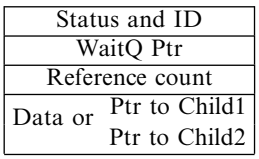
4 Memory management

Memory is divided into two sections, heap space and stack space. We separate the stacks from the heap, since stacks exhibit greater locality and are simpler to manage. This organization, which is similar to that used in the STG-machine, should be contrasted with the organization used in the $\langle v, G\rangle$-machine and SML/NJ where stack frames are allocated in the heap. 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.

[5] We use the term *dereferencing* to mean *removal* of a reference. Operationally, this involves decrementing the reference count and disposing of a term whose count has reached zero.

```
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
```

Fig. 7. Optimized intermediate code for *nfib*.



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 normalized or head-normalized.
Lock: serializes accesses to the node.
Overflow: set if the current node has more than two children.
Type: indicates whether the value of the node is a function symbol, constructor, integer, float, etc.

Fig. 8. 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 with a binary tree of *overflow* nodes containing pointers to reference additional children. Some systems, such as the $\langle v, G\rangle$-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 design's graph nodes is given in figure 8, where each line represents four bytes. The *WaitQ* field points to the notification list of tasks, to be awakened when the (head) normalization of the node is complete. An *ID* field is used for constructors and function symbols, and we record the arity of each node in its status field. The implementation seeks to keep heap nodes small, under the constraints that all heap nodes must be of the same size, and a large reference-count field is required to avoid overflow. Our current design's 20-byte nodes represent

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

an improvement over our original design, which aligned 24-byte nodes on 32-byte boundaries. We optionally allow node alignment in our new design, to reduce false sharing in the cache. However, the performance improvement from alignment has been small and we are currently investigating the matter further.

The locks we have implemented use the *Lock* status bit. They 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 (Sequent Computer Systems, 1987). These locks use the atomic test-and-set (`btsw`) instruction available on the Sequent and generate less bus traffic than naive locks.

The locks themselves are held only as long as it takes to update the status fields. For instance, the system ensures that the *InProcess* bit of a node is set when the graph rooted at this node is under evaluation. When a graph is taken up for evaluation, the evaluator first obtains the *Lock* on the root node. It then checks the *InProcess* bit of the root node, and if set, adds the current task to the *WaitQ*, releases the lock, and suspends the current task. Otherwise, the *InProcess* bit is set, and the lock is immediately relinquished. Similarly, when the evaluation of a graph is complete, the root node is locked only while overwriting the *ID* and the data/child pointers; the lock is released after obtaining a local copy of the *WaitQ* field, so that the suspended tasks may be released to the ready queue *outside* the critical section. Thus, the lock itself is held for very short periods of time.

It may seem as though a status field can be accessed only after the lock is acquired. However, the design permits access to certain status fields without locking, thereby further reducing overheads due to locking. 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 WHNF) 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, in our implementation 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.

It is informative to compare our node design to that of the STG-machine (Peyton Jones, 1992). In particular, our design keeps the *symbol* at the root node of term, whereas the STG-machine stores a code pointer that evaluates the term, even for terms in HNF. In the STG-machine, every access to the term uniformly executes the corresponding code, avoiding the tag checking that EQUALS performs to test whether the term needs evaluation. Due to NF-demand propagation, and the strictness-driven compilation scheme, in EQUALS we find that most terms are already evaluated to the extent needed when they are inspected. Thus the tag checks usually succeed; note

that a successful tag check is cheaper than two context switches that are otherwise necessary in the STG-machine.

Moreover, in the presence of multiple extents of evaluation, we cannot avoid a tag check (or a more expensive lock) in a parallel implementation. In Peyton Jones (1992) it is suggested that the code pointer of a node that is picked up for evaluation be overwritten with a 'queue-me' code pointer such that any subsequent task attempting the evaluation of the same node will get queued. Thus, parallel evaluation may proceed without locking. The important point here is that correctness is not violated if two tasks evaluate the same node and sequence their overwrites arbitrarily, since both evaluations necessarily reduce the term to the same extent. Note, however, that in presence of multiple extents of evaluation, such a scheme may produce incorrect results. For instance, consider a term that is taken up for evaluation (to NF) by a task, say t_{NF}. Before t_{NF} overwrites the WHNF code pointer with pointer to 'queue me' code, let another task, t_{WHNF} take up the same term for evaluation to WHNF. Now, if t_{NF} finishes earlier, the results of the NF computation may be overwritten with the WHNF result computed by t_{WHNF}. In effect, a NF evaluation can never be guaranteed to produce a result in NF! Hence, locking, which may be reduced by using appropriate tags, is unavoidable.

4.2 Heap allocation

To avoid contention when allocating memory, the heap is managed as a two-level structure[6]. 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.

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. Without hysteresis (i.e. empty on zero and full on two), it is possible for an evaluator to cause thrashing at the global pool. For instance, an evaluator that has exactly one full block 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, and this cycle can repeat forever. The hysteresis ensures that each evaluator performs at least n allocations (or deallocations) between any two accesses to the global pool, where n is the number of nodes in a block.

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

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 this reference information is not available for a mark-and-sweep or a copying 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 significant reduction in heap space usage. Furthermore, by maintaining the free list as a LIFO, we encourage immediate reuse of memory that is freed. Since nodes are created and destroyed very quickly in typical programs, this strategy increases the chances of using the same set of memory locations repeatedly, thus improving locality (see section 6).

4.4 Stack management

The stack space is divided into many different stacks, and each task, upon its creation, is allocated one stack. 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 overflow is performed at every entry to a function. When returning after a call that had overflowed, the current stack is unlinked and execution resumes on the old stack.

In the second approach, the stack is expanded on overflow and underflow cannot occur. 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, in theory, be achieved by adjusting the virtual-to-physical address translation tables. This approach incurs some overhead from adjustment, but it has certain advantages relative to linking; for instance, 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, the initial design used the second approach of expanding stacks, to avoid this problem. Unfortunately, on the Sequent Symmetry, the address translation tables are inaccessible to nonprivileged programs, and the operating system interface to them imposes unacceptable limitations on the allowable memory layout. Hence, in the original implementation, we were forced to copy the contents of the smaller stack onto the larger stack. Since this copying process leads to a burst of bus traffic

and can be extremely wasteful of memory (since no 'un-expansion' was ever done), our new design uses the linking approach, and a two-level allocation scheme also manages the new pool of linkable stacks. Kaser *et al.* (1994) contains a further comparison 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 WHNF. Task creation consists of building the term to be evaluated (if it does not already exist), and allocating a task control block, and a stack to be used for the term's evaluation.

Each task's information is stored in a task control block (TCB) that includes not only the task's stack base and limit, but also a field linking the tasks awaiting the same event. The original design imposed a fixed maximum number of tasks, and allocated them from an array. Since the synchronization mechanism used in that design required each graph node indicate the task processing it, these TCB pointers could be shorter (an index) than otherwise possible with full-length pointers. However, the limit on the number of tasks was small, and it sometimes provided a throttle on the creation of tasks. This phenomenon is described later in this section.

5.1.1 Task life-cycle

At the time of creation of a task, an initial stack is linked to a new TCB, and a code pointer and other information are pushed onto the stack. If the task is to normalize an expression $f(a_1,\ldots a_k)$ where f is known at compile time, the initial code pointer indicates the code that evaluates f to the desired extent. The expressions $a_1,\ldots,a_k$ have been appropriately evaluated, and have been pushed onto the new stack as parameters. In the other case, the expression is unknown, and a pointer to it is pushed onto the new stack, where it will be a parameter to the runtime support routine that performs the appropriate normalization. The code pointer pushed onto the new stack thus indicates this runtime routine.

After creation, task T is entered into the global ready queue. Eventually, it will be dequeued, and execution will begin (either initially, or where it was suspended; our mechanism is uniform and requires no tag information). In addition, the information in the TCB sets the evaluator's stack pointer, base, and limit during this execution. While executing, T may create another task S; if so, eventually T will synchronize with S. During this synchronization, S's TCB will be used for parent–child communication, since it is known to both tasks. Specifically, the TCB of S contains a flag that indicates whether S has completed, and a field to hold the return value of S upon completion. If S's completion flag is set, T extracts S's return value.

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

Otherwise, T must suspend itself to await S's completion; this requires that T has the appropriate structures built on its stack and TCB. Later, when S completes, it places T's TCB in the ready queue, ensuring T's re-awakening. Finally, when T is completed, it relinquishes its stack, and writes its return value onto its TCB. It then awakens any other tasks awaiting its completion. It should be noted that the completion flag and the return value fields in a TCB can be overlaid on the stack base and limit fields. Details of the layout of the TCB, as well as a more detailed description of the task-management scheme and performance measurements, can be found in Kaser *et al.* (1994).

5.2 Task synchronization

While evaluating a term t, a task T may find that it must evaluate one of its subterms, say s. Task T may create a new task S (as discussed above) for s, or T may evaluate s by itself. Synchronization is clearly required in the first case, since T may have to wait for the 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 enters it into a wait queue for s. Its evaluator then proceeds to execute the next task from the global ready queue. Note that there is no need to pre-empt 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 $s_1, \ldots, s_k$ 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 $s_2, \ldots, s_k, s$. 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 Hwang and Rushall (1992), where it necessitates additional measures to avoid deadlock.)

Since a term may be evaluated to different extents (WHNF 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 = WHNF: T_2 is added to t's wait queue. Since the code executed evaluating t to NF does not produce an intermediate result in WHNF, T_2 waits until t is in NF. This potentially reduces parallelism, since T_2 is blocked longer than strictly necessary. Nevertheless, efficiency is improved; often, T_2 would soon request subterms of t in WHNF, duplicating the work already being done to evaluate t to NF.

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

- ext_1 = WHNF and ext_2 = NF: T_2 is added to t's wait queue, but will not be released when t's WHNF 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. Although we might release terms awaiting t's WHNF 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 other implementations (Augustsson and Johnsson, 1989; George, 1989; Maranget, 1991). 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 the parent task, say 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, and in certain (probably unlikely) cases work may be duplicated. We do not use this scheme in EQUALS since our experiments indicate that the ready queue is not a bottleneck. Moreover, given the stack and node structures used in EQUALS, this scheme would require that we frequently construct closures, and then suffer the penalty of copying arguments to the stack when entering the closure. Our optimized task creation, whereby the arguments are placed directly onto the newly created task's stack, would not be possible.

5.2.1 Implementation of synchronization mechanism

During its evaluation of some term t, a parent task T may create several child tasks, $S_1, S_2, \ldots, S_k$, to evaluate sub-terms $s_1, s_2, \ldots, s_k$, respectively. Suppose T then needs some of the values being computed before it can proceed further to compute, for instance, $s_2 + s_4$. One approach is to associate a *wait counter* with t (George, 1989). This counter is incremented whenever a child is created, and decremented by each child, upon its completion. We then require the counter to reach zero (i.e. all of $S_1, S_2, \ldots, S_k$ complete) before T proceeds, and this implies that a (conceptual) queue (of length zero or one) of waiting tasks is associated with a wait counter. Each child term e_i then maintains a queue of wait counters to decrement upon completion, allowing shared subterms to be evaluated without duplicating work. To minimize node size, the wait counter can be stored on T's stack, as a local variable in the activation record created for e's normalization. Note the disadvantage of this scheme: the non-required tasks (all but S_2 and S_4 in the above example) cannot continue to run in parallel; with a single wait counter, we cannot selectively wait for some children.

A more flexible approach avoids this disadvantage. It associates a collection of wait counters with e, and each **WaitFor** statement in the code is assigned its own wait counter. Each completing child task will then decrement the counter associated

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

with its value's first use by T, and T can continue execution whenever the wait counter for which it is blocked reaches zero. Thus, T can proceed in parallel with its remaining children. Again, these wait counters can be stored in the activation record for e, and this flexible scheme was used in the initial implementation of EQUALS.

A number of disadvantages lead us to a new mechanism for synchronization. First, it is undesirable to have child tasks making changes in the local stacks of their parents. In particular, the scheme interacted very poorly with stack copying, and significant overhead (and design complexity) was required to ensure that a child task would update the proper counter after an upgrade. Our solution involved having each graph node indicate the task processing it, and thus the nodes were enlarged. A second disadvantage is that it is incorrect to awaken a task just because one of its wait counters has reached zero. Rather, the system must ensure that the task is waiting *and* is waiting for the counter that has just become zero. This requires additional book-keeping, and in our implementation, further enlarged the graph nodes. We also observed that the flexibility did not seem very useful since, in most cases, the wait counters took binary values. This implies that each counter is uniquely identified with a single child, and hence we could let the parent await the child directly, rather than use an intermediate counter. Thus, the new design maintains an explicit wait-queue of TCBs per node. This mechanism is simple and efficiently implemented; however, it can lead to unnecessary awakenings. To see this, suppose that two tasks have been created, and both must be awaited. Using wait queues, two **WaitFor** instructions are required in series, possibly leading to unnecessary synchronization, where a task is awakened only to re-suspend immediately. However, our new design's synchronization overheads are sufficiently low that this situation, even when it occurs, does not seem to degrade our performance significantly.

5.3 Queue management

In EQUALS, new or resumed tasks are placed in the global ready queue from which free evaluators take up tasks. Thus, the global ready queue is the mechanism for load balancing. The 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 described in George (1989).

The queue is viewed as a series of *slots* (see figure 9), 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; if the slot is empty, the evaluator busy waits *on this slot* until it is filled[7]. When a task is placed in the queue, it is placed in the next empty slot. Note that once a slot is assigned, no locking is needed for

[7] Although it is possible to make the idle evaluators relinquish the processor instead of busy-waiting, we have not explored this avenue.

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

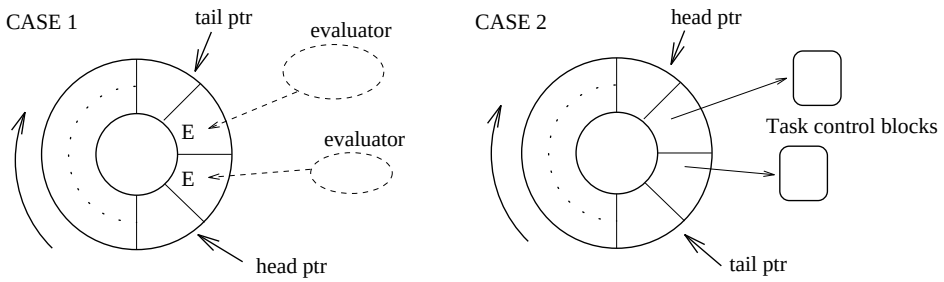


Fig. 9. Split lock queue. Left: No tasks, and two idle evaluators have been assigned to empty slots. Right: Two tasks, and no idle evaluators.

dequeueing. Furthermore, once a slot is assigned to an evaluator, filling other slots does not affect this evaluator's behaviour; hence, evaluators do not race one another. More importantly, the locks needed to enqueue and to dequeue are independent. Such a queue can be easily implemented as a circular array of task pointers. In our implementation, a secondary linked list (with full locking) manages the extremely rare cases when queue overflow occurs (Kaser *et al.*, 1994). The speedup curves shown in section 6 show that this implementation of the ready queue is not a system bottleneck. (An early implementation of EQUALS used a simple linked list of tasks, which proved to be a bottleneck and led to the above improvement.)

5.4 Control of parallelism

A common problem in parallel implementations is that apart from effective techniques to uncover and exploit parallelism, we often need techniques to *control* the available parallelism to match the available resources. Since there may be more parallel tasks than can be efficiently evaluated with the resources (e.g. processors, shared memory) at hand, tasks must be created only when they are deemed useful. In EQUALS, we use two criteria – number of ready tasks as a measure of processor load, and the number of total tasks as a measure of memory load – to measure the total system load. When the system load is high, the evaluators avoid creating tasks and instead perform the intended computation locally. (Two alternative blocks of code will be present at the site of certain function calls. One will spawn a parallel task to perform the computation, and the other performs it locally.) Throttling task creation based on processor load has been used previously (George, 1989; Maranget, 1991).

5.4.1 Thresholds based on processor load

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

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

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 (as controlled by the throttles S_{seq} and S_{par} based on other resources, as explained below). 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, $N_{par} = 1$ and $N_{seq} = 20$ yields good overall results. Note that the flag indicating the system's mode need change 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.

5.4.2 *Thresholds based on memory load*

The number of tasks in the system is an indicator of the extra resource (stack space) usage due to parallelism. Therefore, we base our measurement of the memory load on the total number of tasks in the system. The primary aim of throttling task creation based on the total number of tasks is to avoid pushing the system to a point of failure due to overallocation of resources. As a side-effect, we find that, by forcing increased task granularity, the performance on certain divide-and-conquer programs is improved.

In the original implementation that used stack copying approach to handle overflows, the throttle depended upon the ability of the system to find a small stack for the task (it is inefficient to begin execution on a large stack), and these small stacks were often scarcer than tasks. In a very 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, a large hysteresis was used, involving two thresholds S_{seq} and S_{par}. These thresholds, and the current number of tasks, controlled whether the runtime system would permit creation of new, parallel tasks, or whether it would enforce sequential execution.

In the new design of EQUALS, the situation is simpler. There is no arbitrary fixed limit to the total number of tasks permitted, and those tasks that have completed (but have not yet synchronized with their parents) occupy little memory. Thus, the new design bases its system-wide task throttle only on the number of tasks that have not yet completed. With the new design, our experiments indicate that the throttle is now much less frequently imposed.

5.4.3 Lazy task creation

Note that the technique of throttling task creation based on system load reduces the number of tasks created, and consequently decreases parallel overheads. However, it has been observed by Mohr *et al.* (1991) that this may result in too few tasks being created. In particular, consider a term t where subterms s_1 and s_2 may be evaluated in parallel. At compile time, evaluation of one of the two, say s_2, is sequentialized into the current task, and code is generated for potential parallel evaluation of s_1. At runtime, if the system load is high when the parallel task creation for s_1 is attempted, then even s_1 is evaluated by the parent task. Now, the evaluation of s_1 and s_2 have been irrecoverably sequentialized. The solution proposed in Mohr *et al.* (1991), called *lazy task creation*, is to avoid sequentializing s_2 at compile time, but rather allow s_2 to be evaluated in parallel (i.e. allow s_2 to be *stolen* by another task) when sufficient resources become available.

To make s_2 stealable, we need to create a structure containing all the information necessary to compute s_2 – in effect, a closure for s_2. Note that heap creation of closures can be considerably more expensive than stack creation, and furthermore, synchronization tests are necessary even when these closures are sequentialized. The overheads of lazy task creation will be justified only when the simpler scheme used in EQUALS leads to a severe loss of parallelism. We have, thus far, not observed such a situation.

6 Implementation results and discussion

In this section we describe the sequential and parallel performance of the current implementation of EQUALS[8]. For the results of our initial implementation, see Kaser *et al.* (1992). 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 the speeds and scalability of EQUALS with those of the $\langle v, G\rangle$-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 the sequential performance is competitive with that of modern generational collectors.

6.1 Sequential performance of EQUALS

Table 1 compares the performance of EQUALS with that of SML/NJ (release 0.93), both running on a Sparcstation LX with 32 MB of physical memory. SML/NJ is a sequential implementation of SML, a strict language, and is among the fastest functional language implementations. This comparison is important, since scalability is more easily achieved when one begins with an inefficient sequential implementation. Our sequential implementation is derived from the parallel implementation by

[8] Some of these measurements were previously reported in Kaser *et al.* (1994).

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

Table 1. *Comparison of* EQUALS *and SML/NJ*

	EQUALS	SML/NJ 0.93
Euler	16.5	27.9
Nqueens	28.7	19.2
MatMult	3.4	3.9
Sieve	15.0	8.8
QuickSort	5.6	2.4
FFT	37.3	72.5

Table 2. *Comparison of* EQUALS *with* $\langle v, G\rangle$*-machine and GAML*

	EQUALS	$\langle v, G\rangle$	GAML
Euler	97.1	128.4	430
Nqueens	162.5	73.9	467
Nfib	28.7	62.1	213

simply redefining certain macros. In particular, the sequential implementation does not have to handle locking, stack overflow, task creation or task synchronization.

The example programs for which the performance results are reported in the table are adapted from programs given in Augustsson and Johnsson (1989), Goldberg (1988a), George (1989) and Sekar *et al.* (1990). Among them, *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. The implementation of *Nqueens* is based on the list-of-successes technique discussed in Bird and Wadler (1988). *FFT* computes a waveform, calculates its FFT, and then computes the inverse FFT of the transform.

Observe, from Table 1, 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 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 stores unboxed integers in the heap, and thus integer lists have only half as many nodes as in EQUALS, where all heap objects are boxed by default (see Kaser *et al.* (1994) for a discussion on unboxing in EQUALS). This is not a problem in the first three examples, since the number of steps that access lists or perform other computations are much larger

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

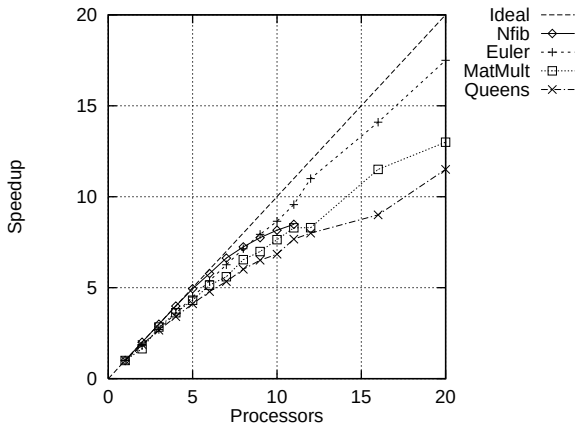


Fig. 10. Speedup curves for EQUALS.

than those that create or destroy lists. For example, in *MatMult* there are 10^6 operations of the first kind versus 10^4 list creation/deletion steps.

6.2 Parallel performance

Table 2 shows wall-clock times for (the parallel version of) EQUALS, the $\langle v, G\rangle$-machine and GAML on a single processor. Timings for both EQUALS and the $\langle v, G\rangle$-machine were obtained on Sequent Symmetries with 16 MHz clocks. However, the $\langle v, G\rangle$-machine timings do not include garbage collection time, which can account for up to 30% of the total (sequential) time. (Although a new concurrent garbage collector appears to reduce this factor, the new timings reported in Röjemo (1991) do not include *Euler, MatMult* or *Nqueens.*)

GAML timings were obtained on a Sequent Balance, which is considerably slower than a Symmetry. This impedes a reasonable comparison between the performance of EQUALS and that of GAML. However, it is mentioned in Maranget (1991) that the sequential execution times for GAML are roughly of the same order as those of the $\langle v, G\rangle$-machine. Timings for EQUALS were obtained primarily on a 12-processor Symmetry at Stony Brook. Timings with 12, 16, and 20 processors (for *Nqueens, Euler*, and *MatMult*) are taken from Kaser *et al.* (1994), and were measured on a 26-processor Symmetry at Argonne National Labs that was available to us only until the Fall of 1994.

Figure 10 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*, EQUALS still scales well, showing that we have managed to keep down task overheads and contention at the global queue.

Figures 11 and 12 compare the scalability of EQUALS with that of the $\langle v, G\rangle$-machine and GAML on *Euler*, *Nfib* and *Nqueens*. Observe that EQUALS scales as well as the $\langle v, G\rangle$-machine and GAML on *Nfib*. On *Euler*, it scales as well as the $\langle v, G\rangle$-machine and better than GAML. It should be noted that the $\langle v, G\rangle$-machine

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

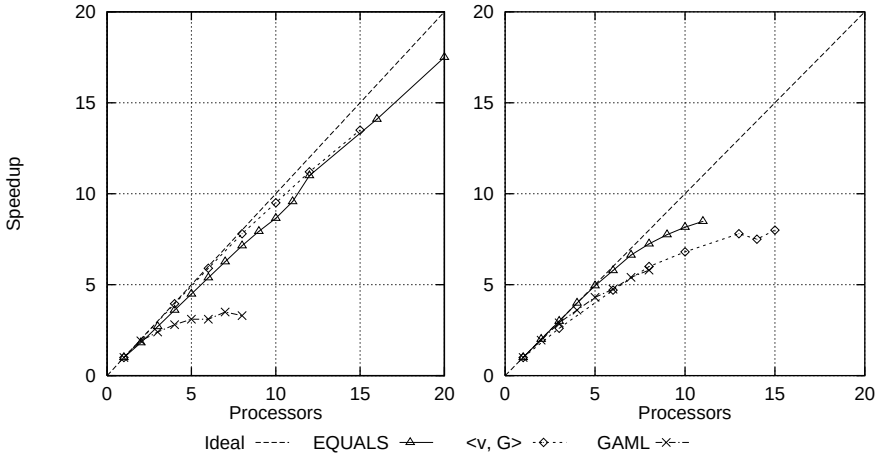


Fig. 11. Speedup curves. Left: *Euler*. Right: *Nfib*.

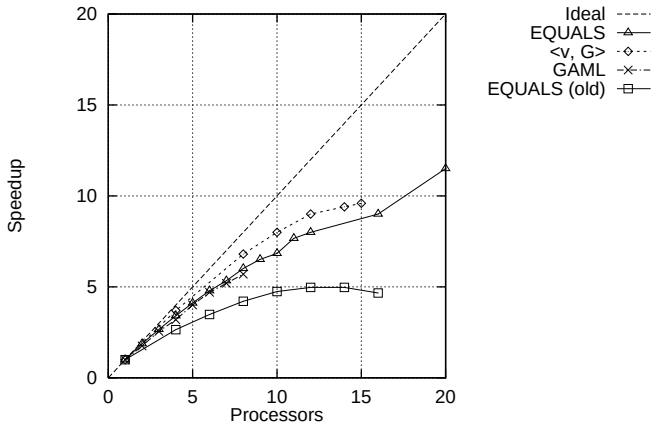


Fig. 12. Speedups for *Nqueens*.

timings do not include garbage-collection times, which may cause its scalability to appear better than in reality: 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 the number of processors increases to 8. In figure 12 we also compare the performance of the current version of EQUALS with the old version presented in Kaser *et al.* (1992) on *Nqueens*[9]. We speculate that this program demonstrates the effect of removing the ceiling on the number of tasks allowed: in Kaser *et al.* (1994) this program was observed to have over 8000 tasks simultaneously alive.

Apart from these small benchmark programs that help measure the performance

[9] Timings for the old version were obtained on a 20-processor Symmetry at Rice University.

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

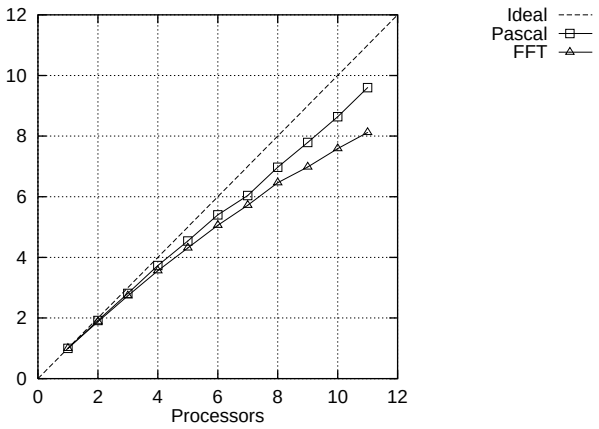


Fig. 13. Speedup curves for EQUALS on two medium-size programs.

of various aspects of the system, we also show the performance of EQUALS on two medium-sized programs in figure 13. *Pascal* is an interpreter for a subset of Pascal, has 83 functions and 833 lines of code, and relies heavily on lazy evaluation[10]. Most of its parallelism arises when a non-strict argument is evaluated in parallel with other strict computations due to run time demand propagation, much like the parallelism found in the *take* example discussed in section 2.3. The other program is *FFT*, which is composed of 43 functions in 343 lines of code. The figure indicates the viability of the EQUALS system in exploiting significant parallelism in moderate-sized applications. It shows that sufficient parallelism can be detected and exploited automatically, reducing the dependency on user annotations.

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.

One program for which efficient memory management is crucial is the *Euler* program. It 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 the EQUALS memory manager. In contrast, the speedup of GAML appears to saturate even for five processors, largely due to poor scaling of the memory management techniques used.

Reference counting collection enables immediate reclamation of free space that leads to low memory requirements. This is illustrated in Table 3, where the memory usage of EQUALS is compared with that of SML/NJ. The *Euler* program cycles

[10] Note that since *Pascal* relies on laziness, it cannot be used to compare the performance of EQUALS and SML/NJ.

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

Table 3. *Memory Utilization of EQUALS vs. SML/NJ (in MB)*

	EQUALS			
	Heap	Stack	Total	SML/NJ
Euler	0.06	0.22	0.28	2.24
Nqueens	1.3	1.1	2.4	2.32
MatMult	0.8	0.02	0.82	1.20
Sieve	0.4	1.1	1.5	1.85
QuickSort	0.20	0.56	0.76	0.95

through memory and the table shows the substantial gains that can be obtained via immediate reclamation. EQUALS consumes more memory than SML/NJ on *Nqueens*, primarily due to the boxed representation of integers in the heap. This effect is also seen on *Sieve* and *QuickSort*, where the difference between the memory consumption of EQUALS and SML/NJ is small. However, it should be noted that SML/NJ allocates stack frames from the heap, and hence there is no separation of the memory area. Since references to the stack typically show much greater locality than heap references, and moreover, since stacks are considerably easier to manage, stack usage is less critical to overall system performance.

It has been commonly thought that reference counting leads to higher memory consumption due to the presence of reference count fields in every heap node. Although it is true that heap node sizes do increase due to reference counting, due to the potential for immediate reclamation of dead heap, we observe an overall reduction in the amount of heap used.

It is sometimes stated that reference counting damages locality because the simple removal of a pointer requires that the pointed-to object be modified in its count field. Since modern generational garbage collectors (such as employed by SML/NJ) have a reputation for enhancing locality, we measured the performance of EQUALS and SML/NJ as we gradually reduced the amount of physical memory available from about 21.5 MB to 2.5 MB. To do this, we used a Sparcstation-LX with 32 MB of physical memory. Under Solaris 2, it is possible to effectively disable portions of physical memory. The machine was removed from the network and all non-essential processes, such as the windowing system and the various daemons, were killed. In this state, the machine had nearly 21.5 MB of physical memory available for user processes. Varying amounts of this available memory were disabled, and the SML and EQUALS versions of each example program were run to determine the effect of the memory shortage. Since our mechanism to directly measure paging activity itself created additional paging, we used elapsed times instead as a measure of the paging activity. Moreover, to ensure paging for both EQUALS and SML/NJ[11], we timed the

[11] The heap-size parameters used were defaults supplied with the distribution of SML/NJ 0.93. This begins with 5 MB heap, but our examples caused ML to dynamically expand the heap beyond this.

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

Table 4. *Runtimes of* EQUALS *and SML/NJ 0.93 on larger problem instances*

	EQUALS	SML/NJ
Euler(2000)	68.7	124.0
Nqueens(11)	174.0	104.2
MatMult(250)	49.7	59.6
Sieve(40000)	153.8	107.9
QuickSort(20000)	32.1	12.7

Nqueens benchmark on an 11×11 board, the *MatMult* benchmark with 250×250 matrices, *Euler* and *Sieve* with inputs 2000 and 40,000 respectively, and *QuickSort* with 20,000 elements.

To isolate the time due to paging, we then subtracted the wall-clock times obtained from the times observed when all physical memory was available. (Ten runs were averaged to obtain each data point.) Results are shown in figure 14. Thus, while generational collectors have an excellent reputation for enhancing locality, our results indicate that a reference-counting collector can be a viable alternative.

6.3.1 Cost of reference counting

Finally, we quantify the cost of reference counting in our implementation. Two issues are involved, since reference counting has a direct cost associated with the actual reference manipulations and their cache effects. It also has an indirect benefit, due to the enhanced locality from immediate reclamation. Below, we describe the experiments and the measurements, obtained using a Sparc-20 with 64 MB of physical memory.

We estimated the total penalty of reference counting in EQUALS by measuring the performance improvement due to the removal of all reference count manipulations. Since in the absence of reference information, memory is never deallocated, allocation was done from a pre-initialized, consecutive freelist. Note that although the modified system has no direct overheads due to absence of reference count manipulations, it can suffer from cache degradation since memory is never reclaimed. Using the modified system, we observed an average speed-up of 14% over the base system.

To separate the cost from the benefit, the programs were timed when counts were manipulated as usual, but freed nodes were discarded, rather than added back to the freelist. This version incurs all of the direct costs of reference counting, but allocates the same set of nodes as the first experiment. Therefore, it should similarly lack the locality benefits due to immediate node reuse. We observed a 26% slowdown compared to the first experiment. This slowdown, we believe, approximates the total cost due to reference counting. Note that the modified system exhibits a slowdown of 11% over the base system, and this approximates the benefit due to the locality of reference enabled by immediate memory reclamation. Thus, the above experiments indicate that the cost of the reference counting operations is approximately halfway offset by their locality benefit.

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

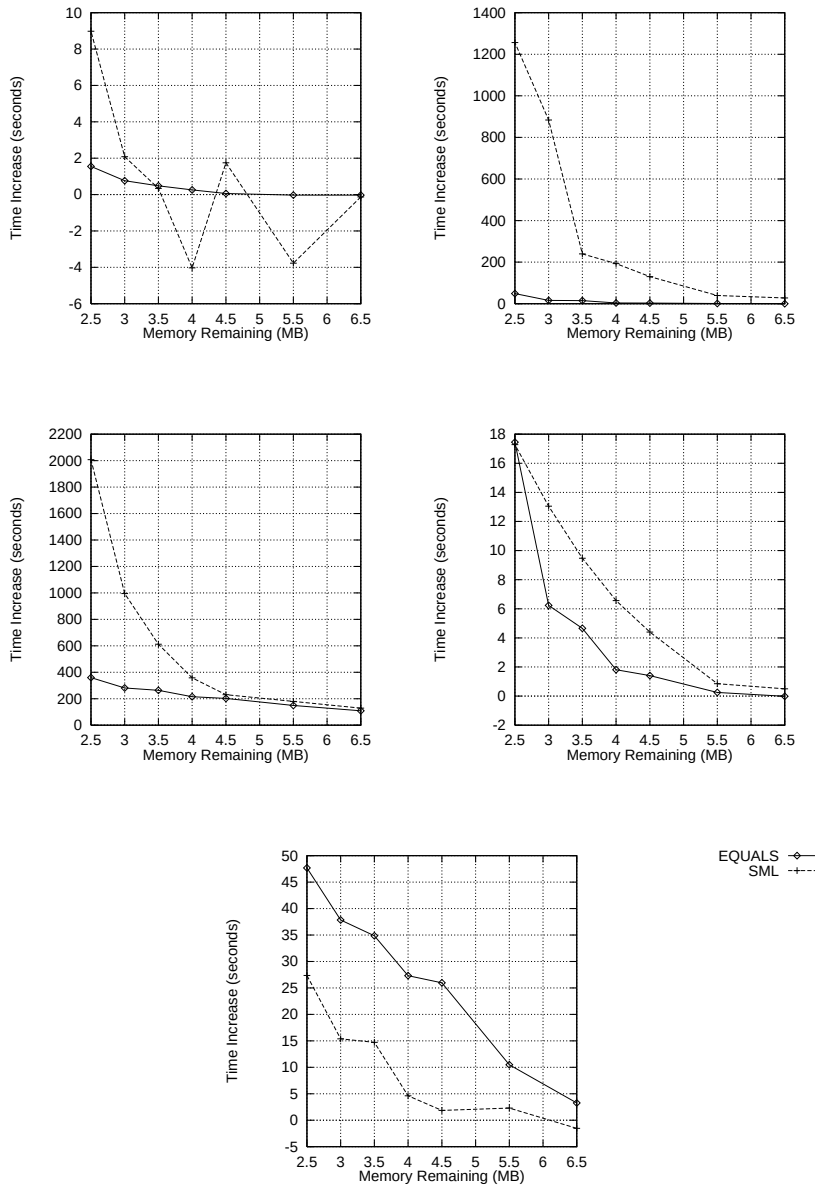


Fig. 14. Time increases when physical memory is limited. Top: *Euler* (left) and *MatMult* (right). Middle: *Nqueens* (left) and *QuickSort* (right). Bottom: *Sieve.*

7 Concluding remarks

The EQUALS implementation demonstrates that it is possible to automatically detect and effectively exploit the parallelism implicit in functional programs by propagating NF-demand. Strictness-based propagation of NF-demand means that the advantages of NF evaluation can be safely performed in presence of laziness. In addition, run-

https://doi.org/10.1017/S0956796897002669 Published online by Cambridge University Press

time propagation of NF-demand leads to discovery of more parallelism than is possible with static propagation alone. Code for NF evaluation is very similar to the code generated for strict languages. This possibly explains why the uniprocessor performance of EQUALS is comparable to that of strict implementations such as SML/NJ.

In addition to the benefits of NF-demand propagation, the EQUALS implementation indicates the viability of reference counting as an effective technique for memory management. Our experiments with EQUALS demonstrate that reference counting not only scales very well, but it also has good sequential performance. Moreover, use of reference counting leads to better memory utilization and exhibits good locality.

The two main features of EQUALS, namely, the use of NF-demand propagation and reference counting enable various further optimizations. For instance, the load balancing scheme used in EQUALS is quite simple, and its power can be considerably improved by using compile-time estimates of the time complexity of functions. Static analysis of time complexity, as well as mechanisms to maintain the size information at run-time, are considerably simpler in the case of NF evaluation. With regard to reference counting, the reference information enables us to avoid updating closures that are not shared; note that such a run-time technique is necessarily more effective than any technique based on static sharing analysis. The impact of these optimizations on the performance, however, remains to be quantified.

Acknowledgements

We thank the referees for their detailed comments that led to substantial improvement of the paper. This research was supported in part by grants from the National Science Foundation (CCR-9404921, CDA-9303181, CDA-9504275 and INT-9314412) and the NSERC of Canada (OGP0155967). Use of a Sequent Symmetry S81 was provided by the Department of Computer Science at Rice University under NSF Grant CDA-8619393, and another by the Mathematics and Computer Science Division of Argonne National Laboratory, which is operated by the University of Chicago under a contract with the U.S. Department of Energy.

References

Appel, A., Ellis, J. and Li, K. (1988) Real-time concurrent collection on stock multiprocessors. *ACM Symposium on Programming Language Design and Implementation*, pp. 11–20. ACM Press.

Arsac, J. and Kodratoff, Y. (1982) Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, **4**(2), 295–322.

Augustsson, L. (1984) A compiler for lazy ML. *ACM Symposium on Lisp and Functional Programming*, pp. 218–227. ACM Press.

Augustsson, L. and Johnsson, T. (1989). Parallel graph reduction with the $\langle v, G \rangle$ machine. *Symposium on Functional Programming Languages and Computer Architecture*, pp. 202–213. ACM Press.

Bird, R. and Wadler, P. (1988) *Introduction to Functional Programming*. Prentice Hall.

Darlington, J. and Reeve, M. (1981) Alice: A multi-processor reduction engine for the parallel evaluation of applicative languages. *Symposium on Functional Programming Languages and Computer Architecture*, pp. 65–75. ACM Press.

George, L. (1989) An abstract machine for parallel graph reduction. *Symposium on Functional Programming Languages and Computer Architecture*, pp. 214–227. ACM Press.

Goldberg, B. (1988a) Buckwheat: Graph reduction on shared-memory multiprocessor. *ACM Symposium on Lisp and Functional Programming*, pp. 40–51. ACM Press.

Goldberg, B. (1988b) *Multiprocessor execution of functional programs.* PhD thesis, Yale University.

Hudak, P. (1987) A semantic model for reference counting and its abstraction. In: Abramsky, S. and Hankin, C. (eds.), *Abstract Interpretation of Declarative Languages*, pp. 45–62. Ellis Horwood.

Huelsbergen, L. and Larus, J. (1993). A concurrent copying garbage collector for languages that distinguish immutable data. *Principles and Practice of Parallel Programming*, pp. 73–82. ACM Press.

Huet, G. and Levy, J.-J. (1991) Computation in orthogonal rewriting systems. In: Lassez, J.-L. and Plotkin, G. (eds.), *Essays in Computational Logic – Essays in Honor of Alan Robinson.* MIT Press.

Hughes, R. J. M. (1982) *Reference-counting with circular structures in virtual memory applicative systems.* Technical Report, Programming Research Group, Oxford.

Hwang, S. and Rushall, D. (1992) The *v*-STG machine: A parallelized spineless tagless graph reduction machine in a distributed memory architecture. *Fourth Workshop on Parallel Implementations of Functional Languages.*

Kaser, O., Pawagi, S., Ramakrishnan, C. R., Ramakrishnan, I. V. and Sekar, R. C. (1992) Fast parallel implementation of lazy languages – the EQUALS experience. *ACM Symposium on Lisp and Functional Programming*, pp. 335–344. ACM Press.

Kaser, O., Ramakrishnan, C. R. and Sekar, R. C. (1994) A high performance runtime system for parallel evaluation of lazy languages. *First International Symposium on Parallel Symbolic Computation*, pp. 234–243. World Scientific.

Laville, A. (1988) Implementation of lazy pattern matching algorithms. *European Symposium on Programming: Lecture Notes in Computer Science 300*, pp. 298–316. Springer-Verlag.

Maranget, L. (1991) GAML: A parallel implementation of lazy ML. *Symposium on Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 523*, pp. 102–123. Springer-Verlag.

Mohr, E., Kranz, D. and Halstead, R. (1991) Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, **2**(3), 264–280.

Park, Y. and Goldberg, B. (1995) Static analysis for optimizing reference counting. *Information Processing Letters*, **55**(4), 229–234.

Peyton Jones, S. L. (1992) Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, **2**(2), 127–202.

Peyton Jones, S. L. and Launchbury, J. (1991) Unboxed values as first class citizens in a non-strict functional language. *Symposium on Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 523*, pp. 636–666. Springer-Verlag.

Peyton Jones, S. L., Clack, C., Salkild, J. and Hardie, M. (1987) GRIP – a high-performance architecture for parallel graph reduction. *Symposium on Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 274*, pp. 98–112. Springer-Verlag.

Puel, L., & Suarez, A. (1990). Compiling pattern matching by term decomposition. *ACM Symposium on Lisp and Functional Programming*, pp. 273–281. ACM Press.

Röjemo, N. (1991) *A concurrent garbage collector for the* $\langle v, G \rangle$*-machine*. Technical Report, Chalmers University.

Sekar, R. C., Pawagi, S. and Ramakrishnan, I. V. (1990) Small domains spell fast strictness analysis. *ACM Symposium on Principles of Programming Languages*, pp. 169–183. ACM Press.

Sekar, R. C., Ramesh, R. and Ramakrishnan, I. V. (1992) Adaptive pattern matching. *International Colloquium on Automata, Languages and Programming: Lecture Notes in Computer Science 623*, pp. 247–260. Springer-Verlag.

Sequent Computer Systems (1987) *Sequent guide to parallel programming*.

Watson, P. and Watson, I. (1987) Evaluating functional programs on the FLAGSHIP machine. *Symposium on Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 274*, pp. 80–97. Springer-Verlag.