Proceedings of PASCO'94

A High Performance Runtime System for Parallel Evaluation of Lazy Languages

Owen Kaser*
Dept. of MSCS,
UNB at Saint John,
Saint John, NB, Canada
owen@unbsj.ca

C.R. Ramakrishnan†
Dept. of Comp. Science,
SUNY at Stony Brook,
Stony Brook, NY 11794
cram@cs.sunysb.edu

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

Abstract: is a system for parallel evaluation of lazy functional programs implemented on a Sequent Symmetry. The distinguishing features of over previous implementations are propagation of Normal Form demand and memory reclamation via reference counting. In an earlier paper, the validity of these ideas was established based on a preliminary implementation. Our experience, based on extensive experimentation with this implementation, led to the design and implementation of a new high-performance runtime system for described in this paper. We discuss the rationale behind the various design choices and provide quantitative evidence for the validity of these choices.

Keywords: Lazy Functional Languages, Implementation, Parallelism, Performance Evaluation

1 Introduction

Functional languages offer a conceptually simple approach to programming parallel computers. Detection and exploitation of parallelism in functional programs is simpler than in imperative programs, due to referential transparency. To implement functional programming systems on parallel machines, efficient implementation of the runtime system is crucial for the overall system performance. This system manages tasks and associated resources, such as memory, and this paper describes our experience with the design, implementation and refinement of a high performance runtime system for EQUALS [5].

EQUALS is a system for parallel execution of lazy functional programs, implemented on the Sequent Symmetry. The distinguishing features of EQUALS over previous implementations on shared memory machines (e.g., Buckwheat [3], GAML [6], and the $\langle \nu, G \rangle$-machine [1]) are the propagation of exhaustive Normal Form (NF) demand in addition to the traditional Weak Head Normal Form (WHNF) demand, and use of reference counting to reclaim memory. In an earlier paper [5], we established the validity of these ideas based on a preliminary implementation. Since then, extensive experiments have been conducted on the system exposing the limitations and bottlenecks in the implementation, and leading to the design of a new high-performance runtime system to overcome these limitations. The design of this runtime system is described in detail in this paper, providing the rationale behind the various design choices.

1.1 Design Objectives

The objective of the new runtime system is to support a scalable memory manager as well as a simple and efficient task manager. Below we briefly summarize our approach towards meeting these objectives.

In EQUALS, memory is divided into separate

*Supported by NSERC grant OGP0155967.
†Supported in part by the NSF (CCR-9102159, CDA-9303181 and INT-9314412).
‡This work was performed while R. C. Sekar was at Stony Brook and supported in part by NSF CCR-9102159.

heap and stack spaces, since the stack and heap display very different characteristics in terms of allocation, access and deallocation. Note that the STG-machine [7] also separates heap from stack, for these reasons. In contrast, the $\langle\nu, G\rangle$-machine stores stack frames in the heap and uses garbage collection to reclaim the heap. This, on the one hand, can lead to poor locality of reference and cycling though the memory. On the other hand, in a parallel setting, the mark-and-sweep and copying collection techniques have inherently sequential components, such as the need to lock every structure before moving, and hence do not scale well.

Reclaiming memory by reference counting is highly distributed and shows little parallel overhead, and is used in EQUALS. Furthermore, to avoid heap fragmentation, instead of using an expensive compaction operation, we build graphs and closures in the heap using nodes of fixed size. Thus using a separate stack space and a heap of fixed-sized nodes, memory allocation and reclamation become inexpensive in both parallel and sequential situations. However, accessing fields in a closure built out of fixed-size nodes may be more expensive than in flat representations, such as those used in TIM [9] and the STG-machine. Nevertheless, propagation of NF-demand in EQUALS reduces the number of closures that are constructed and eventually, but not immediately, evaluated. Implementation results indicate that the additional cost of closure access is small in practice, and is worth the return in terms of lower parallel overheads.

The EQUALS system exploits *conservative* parallelism, in which only needed computations are performed. In contrast, in systems that exploit *speculative* parallelism, for instance the $\langle\nu, G\rangle$-machine and GAML, the results of certain computations may not be eventually used. Although this method may uncover more parallelism, in order to ensure that the needed tasks always make progress, it requires preemption of not only the processor (as proposed in [1]) but all other resources, such as memory, as well. The conservative approach, on the other hand, needs no preemptive scheduling, and results in simpler and more efficient task management.

1.2 Implementation Overview

The EQUALS system consists of a compiler and a runtime support system. The compiler uses *ee*-strictness analysis, developed in [8], to detect parallelism and translates the source program to C via a combinator-based intermediate language. The compiled code is executed under the control of a runtime system that provides facilities for task creation, switching and synchronization. The runtime system also manages the resources, such as memory, needed to execute a task.

The goal of compiled code is to normalize a given input expression. The EQUALS compiler generates two versions of code for each function, which are invoked under WHNF and NF demand respectively. Partial applications (closures), delayed evaluations (thunks) and head normal forms are all represented uniformly as graphs (or expressions; we use the two interchangeably) in the heap. Once evaluated, a graph is overwritten with its normal form, thus sharing the normal-form computation. Since allocation and freeing of stack frames is much cheaper than heap nodes, whenever the root of a graph to be evaluated is known at compile time, the code for this function is directly invoked. Note that this amounts to building and entering the corresponding closure on the stack. The structure of the heap nodes, and their allocation and deallocation mechanisms are described in section 2. The management of stack space, including stack allocation and stack overflow management are discussed in section 3.

If multiple subexpressions need to be evaluated in order to normalize an input expression, these subexpressions can be evaluated in parallel. Subexpressions that are evaluated in parallel are taken up by individual tasks, which execute the compiled code on their private stacks. Since there may potentially 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 might be executed as UNIX processes, it is too expensive to do so. Instead, 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, which is used to share and balance the system load. Given an input expression, a task to normalize this expression is created and placed in the ready queue. Whenever idle, the evaluator processes pick up tasks from the ready queue and start their execution. If a task needs the value of a subexpression that is

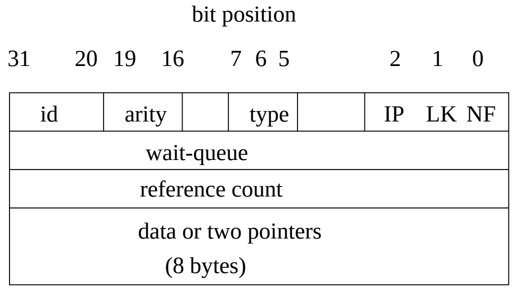

Figure 1: EQUALS node structure.

currently being evaluated by another task, then the first task is suspended, awaiting the subexpression's evaluation. The evaluator then begins execution of a task from the ready queue. Once an expression's evaluation is complete, all tasks awaiting its evaluation are released to the queue.

Mechanisms for task creation, synchronization and switching are described in section 4. In section 5, efficient manipulation of the ready queue, and the use of the ready queue in balancing of system load are discussed. The experimental results presented in section 6 provide evidence for the effectiveness of the design.

2 Heap Management

We first present the design of heap nodes. Our design balances the conflicting requirements of keeping the node size small (and hence reducing the space wasted in representing small objects such as boxed characters) and minimizing the indirection needed to access parts of a large object. We then describe the implementation of efficient and highly parallel schemes for node allocation and deallocation.

2.1 Node Design

Figure 1 shows the layout of a node, which occupies 20 bytes of memory[1]. Its *type* field specifies whether the node is a functor, constructor, integer, floating-point value, or so forth. The *NF* bit is set if the node is the root of an expression in normal form; the *IP* bit is set if the expression is in the process of being normalized; and the *LK* bit is used to obtain exclusive access to the node. The *waitq* field is used for task synchronization and the *refcount* field maintains the reference count for the node. A 32-bit field makes it unnecessary to check for overflow, and the current scheme uses locked increment and decrement instructions to manipulate the counts.

The *id* field holds the identity of the root functor, constructor or partial application; a table maps these values to two code entry points that compute the NF and WHNF respectively. Though the code pointers could themselves have been stored in the node as done in the STG-machine, we would have needed two fields for the two demand types. We chose indirection to minimize the size of the node.

The last two words in the node are used to store either an integer or floating-point value, or up to two pointers. If the node has one or two children, the pointers directly refer to the children. If the node has more than three children, these pointers indicate one or two *overflow nodes* containing pointers to the children. Each overflow node may contain up to 4 child pointers[2]. Note that this one-level tree scheme minimizes the indirection needed to access each child, compared with a linked list of overflow nodes, as used in the earlier EQUALS implementation. Overflow nodes are the same size as graph nodes and are allocated from the same pool. However, we need not use reference counting, since they are an extension of their owners and cannot be shared.

Locking In EQUALS, both high- and low-level locks are used. Low-level locks use the *LK* bit present in each node, and are manipulated with the atomic bit-test-and-set instruction supported by the Symmetry. These locks prevent us from examining a node when it is in an inconsistent state, and are used to provide mutual exclusion from critical sections in the synchronization process. For instance, the type information and the waitq fields can be reliably read or manipulated only when the lock is held. Nevertheless, the design attempts to reduce the amount of locking needed. For instance, the *NF* bit can always be examined without holding the lock. If this bit is set, the implementation guarantees that the node is stable and all fields can be read safely without locking. Determining whether a node represents an expression in WHNF is optimized similarly.

The system also implements high-level locking, ensuring that graphs are not evaluated twice. The in-process bit, *IP*, is used to prevent two tasks from simultaneously normalizing the graph. Finally, for efficiency the reference count field is manipulated using atomic instructions, independent of the node's lock.

[1] The preliminary implementation had 32-byte nodes and still used much less memory than SML/NJ (see [5]).

[2] We assume use of a source-level transformation to limit the arity of each node to 8.

Unboxed Integers in the Heap The EQUALS system optionally supports unboxing of certain primitive types (currently only integers) in the heap. With this representation, the system handles two kinds of integers: boxed integers, whose values are stored in the heap as explained above; and unboxed integers, where a heap pointer to a boxed integers is replaced by a tagged representation of the actual data value. In our representation, all integers are 31-bit values with 0 in the least-significant bit (lsb). All pointers have a 1 in the lsb; each pointer access uses the immediate offset (-1), which has no additional cost in the Symmetry. This enables addition and subtraction of integers without tag manipulation, though other operations need to handle tags.

2.2 Node Allocation and Disposal

Heap nodes are allocated out of a two-level free list that minimizes contention during allocation while maintaining an even distribution of free nodes. The top level is a global pool of *blocks*, where each block itself is a list of free nodes, and has a fixed size, B. Access to the global pool is controlled by a lock. The unit of transaction with the global pool (allocation and freeing) is an entire block.

Each evaluator maintains a private list of free nodes. Nodes are allocated and returned to this free list. When an evaluator holds more than a preset number of free nodes, it builds a block that is returned to the global pool. When an evaluator runs out of nodes, it obtains a block from the global pool. When the global pool is empty, the pool is expanded by dynamically allocating shared memory.

This scheme permits fast allocation with low parallel overheads. For instance, allocating $10k$ nodes consecutively takes only 4.8 μs per node on one processor and 5.1 μs when eight processors perform simultaneous allocation, and the system scales well even on programs that allocate very large numbers of nodes. In contrast, the previous implementation did not have a global pool; rather, UNIX signals were used to force redistribution of free lists amongst evaluators. That implementation prevented the *Euler* example, for instance, from scaling well beyond eight processors.

Disposal When the reference count of a graph node falls to zero, its disposal is triggered. This entails not only deallocating the node (releasing it to a free list), but also deallocating its associated arity-overflow nodes. Furthermore, its children should be dereferenced, and this may in turn lead to their disposal. Therefore, the most straightforward implementation has two mutually recursive routines. However, we often observe large data object being disposed at once. For efficiency, the recursive traversal of the object has been optimized by applying the standard techniques of unfolding a few calls and converting the resulting tail-recursion into jumps in the disposal routine. This improved the per-node time to dispose of a list of lists, from 22 μs per node to 14 μs.

Impact of Unboxing When unboxed values are manipulated, every pointer must have its tag examined before it is used, unless the type of the object is known at compile time. Since the reference increment and decrement instructions operate in an untyped environment, they need to first perform a tag check. The disposal time, which is 14 μs when handling only boxed structures, rises to 18 μs when unboxed structures must be detected. Whether unboxing is worthwhile is thus application dependent, as determining factors include the prevalence of those types that can be unboxed.

3 Stack Management

Stacks, like heap nodes, are a resource managed by the runtime system. Important design considerations for our system's stack management include the sizes of stacks, the allocation technique used, and the support of stack-intensive tasks. For stack size, large stacks promote efficient, contiguous allocation of activation records. However, there can be many suspended tasks at any time, and it would be wasteful if each tied up a large stack when a small one would suffice. Therefore, we must balance these needs, allocating moderately small[3] stacks that can hold a few hundred activation records.

Stack Allocation As with heap nodes, stacks are allocated from a two-level structure with private free lists and a shared pool of stacks. Stacks are allocated either on creation of a new task, or when an existing task has outgrown its current stack. Correspondingly, stacks are freed either when the task completes, or as part of the stack-growth process. Note that stack allocation and deallocation are as simple as heap allocation, and thus as fast.

[3] $4kb$ in our current implementation; $16kb$ in the previous.

Supporting Stack-Intensive Tasks During its lifetime, a task may perform many recursive calls, or invoke a runtime service routine that recursively traverses a large structure. In either case, the small stack initially provided may not be sufficient. Hence, checks (which only involve a comparison and branch) are made at critical points in the compiled code. For efficiency, we guarantee that the first call to a support routine will not overflow the stack, by reserving some space in each stack as a safety margin[4]. Support routines themselves (especially the recursive ones used for deallocating, printing, and normalizing graphs) check for stack overflow, before making a call.

To detect overflow, the compiled code may check either at the beginning of each function, or before every call. Since there are many more call sites than entry points, checking for overflow at entry to every function would result in smaller code size. This was done in our previous implementation. However, after entry to a procedure, the parameters are present in the old stack and they need to be moved to the new stack. This adds overhead. Therefore, it is advantageous to check for overflow before making a call, so that parameters can be pushed on the new stack, and this is what our current implementation does.

When stack overflow is imminent, more stack space needs to be allocated for the task. This may be accomplished by *linking* a new stack to the current stack or by *upgrading* the current stack to a new, larger one. Linking is used in the current implementation, and the reasons for this choice are explained below.

3.1 Stack Linking

With this method, when a stack overflows, a fresh stack is allocated and linked to the existing stack. As the stack requirements of the task diminish, the linked stacks are deallocated. In the current implementation, all stacks are contiguous blocks of memory of fixed size. If, before calling a function, we detect that the current stack has insufficient space, a new stack is allocated, and the current stack pointer is saved in a link field of the new stack. The stack pointer is then made to point to the new stack, but the frame pointer is not affected. Parameters are then pushed for the call (on the new stack), and the call is made. When control returns, the saved stack pointer is restored, and the previously linked stack is deallocated. Overflow handling using this scheme takes $26\mu s$ per overflow, including stack allocation and linking at call time, as well as unlinking and deallocation at return time.

When each call `f(<args>)` is compiled to the form

```
if (oflow) { link; f(<args>); unlink; }
else f(<args>)
```

the overhead for the linking scheme is very low since we need not explicitly test for stack underflow after each return, i.e., whether an unlink is required. A call that does not overflow takes about $9\mu s$ in the above scheme. If a call is compiled such that underflow checks are performed before each return, a non-overflowing call takes about $12\mu s$; as expected, there is no difference for handling overflows. For programs which perform many function calls, this improvement is substantial; for instance, the execution time of *Tak* is reduced by about 20%.

One difficulty with the stack linking scheme is the possibility of repeated stack linking and unlinking. Consider a function that calls several other functions in sequence, and suppose that the stack is about to overflow. Each of these calls would overflow the stack and result in stack linking. On return from each of these functions, these stacks would be unlinked. This effect can be seen on evaluation of the n-queens program. We find that a contiguous $1Mb$-long stack is sufficient for evaluation of $queens(9)$ on one processor, but the linking scheme with $4kb$ stacks performs 33,000 links and unlinks!

3.2 Stack Upgrading

The stack-upgrading approach has been used in previous versions of EQUALS, where a task has a single contiguous stack that can be upgraded to a larger contiguous stack upon overflow. The contents of the smaller stack are copied onto the larger one, the frame pointers in the stack are adjusted, the smaller stack is deallocated, and then execution resumes on the larger stack. This scheme does not exhibit the thrashing behavior of the linking scheme; its obvious drawback is the overhead created by copying. However, we find that stack upgrading itself is rare and hence copying penalties are often very low. For example, in $queens(9)$ on 8 processors, only $143kb$ are copied over all the upgrades.

Our experience with copying showed that the main disadvantage of the stack upgrading scheme is *not* the copying overhead, but its excessive space usage. Note that since a stack is never *downgraded*, the stack requirements of a task may

[4] Currently, 1600 bytes is used as the safety margin, so that standard C library functions may be invoked from a task.

Word 1:	Stack Limit / Return Value
2:	Stack Pointer
3:	Waiting Task Pointer
4:	Stack Base / Completion Flag

Figure 2: Structure of Task Control Block

have shrunk without correspondingly shrinking the stack size. In one example (*Euler*), the upgrading scheme needs more than $15Mb$ of memory, whereas with the linking scheme, at most 30 stacks are used at any given time. However, there are programs for which copying results in a lower stack requirement that linking; note that there is one (often unused) safety margin per stack. With copying there is only one stack per task, whereas with linking the number of stacks per task is unbounded. Nevertheless, this waste in the linking scheme can be reduced by carefully selecting the size of stacks and the safety margin.

4 Task Management

The purpose of a task is to evaluate a graph to either NF or WHNF. Each task is associated with a *task control block* (TCB). After discussing the TCB's structure (figure 2), we present schemes for task creation, switching and synchronization and their implementation.

Task Control Blocks The TCB holds the base and limit of a task's current stack, and for each suspended task, the current top of stack. A task can have at most one task, its parent, awaiting the results of the computation. Computations are shared via shared graphs, as explained later in this section. If the parent waits for the task's computation to complete, a pointer to the parent's TCB is placed in the 'Waiting Task Pointer' field of the child's TCB. Since the child's TCB is accessible from both the parent and the child, the TCB is used to transmit the return value to the parent task. The validity of the return value is indicated by a completion flag in the TCB. Once a task has completed, all its resources except the TCB are released. Since a completed task has no stack, fields are re-used to store the completion flag and the return value. The parent releases the TCB after obtaining the return value.

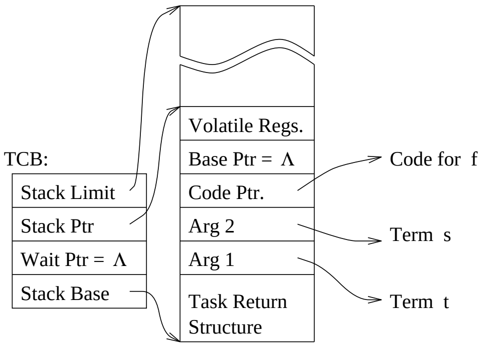

Figure 3: Task Structure created to evaluate $f(t, s)$.

4.1 Task Creation

To create a task, a TCB is first allocated, along with an associated stack whose base and limit are recorded in the TCB. Next, a *task return structure* is placed on the stack. This structure will transfer control to the runtime system, which saves the return value in the TCB and releases the stack. If the task is to evaluate an expression e whose root is a known function, the arguments to the function are then pushed on the stack, followed by dummy initializations for the volatile registers and the pointer to the code that evaluates e. The stack pointer is then saved in the TCB. Likewise, if the task is to evaluate an unknown expression, the task created will invoke the suitable runtime support routine with the expression as its argument. The task structures created, to evaluate a function $f(t, s)$ where t and s are pointers to graphs, is given in figure 3 .

4.2 Task Switching

When a task, say T, suspends awaiting the completion of another task, a *suspension* is created for T. This involves pushing, in order, the program counter, the volatile registers and the base (frame) pointer on the stack. The stack pointer is then saved in T's TCB.

A suspended task is taken up for execution by *entering* its suspension: first restoring the stack pointer from the TCB, restoring the volatile registers and the frame pointer from the stack, and then simply executing a `return` instruction. Note that the structure of a newly created task is the same as that of a suspension, since the stack contains volatile registers, code pointer and values needed by this code in the same order. Hence, every task in the ready queue, either freshly created or recently awakened, is treated uniformly.

Whenever a task is taken up for execution, a structure much like a suspension is created for the evaluator process. The only difference arises because the evaluator process does not have a TCB. Hence, the stack pointer is kept in a fixed location, private to the process. When a task is completed, the suspension for the evaluator process is entered, thus switching to the evaluator process. The evaluator then polls the ready queue for runnable tasks.

Creating a new parallel task and enqueuing it takes only $48\mu s$. Creating a new parallel task that returns immediately, and then awaiting it requires $125\mu s$ if an idle processor is available. Otherwise, the parent needs to be suspended and reinvoked, and $150\mu s$ is typical.

4.3 Task Synchronization

During its processing, a task T may require the evaluation of some graph s. T may either choose to evaluate s itself, or it may wish to create a task. In either case, it must first check that another task is not currently evaluating s, to avoid duplicating work. If another task evaluates s, T must eventually suspend itself to await the evaluation of s. The evaluator then dequeues another task from the global ready queue, and executes it. When s is finally evaluated, its task will awaken all tasks in the wait queue for s, including T. A complication arises from supporting multiple extents of normalization; suppose that T requires NF and s is being processed to WHNF, or viceversa. To simplify the implementation, there is only a single "in-process" status bit and a single wait queue; in fact, there is no way to tell to which extent s is being evaluated. When T awakens, if it is awaiting WHNF it proceeds immediately (since this is the minimum extent); if it is awaiting NF it will check the current extent, and if the graph is not in NF it re-attempts to evaluate it. Possibly, another task T' will may have begun re-evaluating s, and T may again wait. Despite appearances, this is efficient because T' *must* be evaluating s to NF, and T will never have to wait again.

Synchronization and Sharing A wait queue is associated with each graph node, since a graph node and its ancestors may be processed by the same task. Consider a task T created to evaluate an expression s; it may also take up many subexpressions $s_1, ..., s_k$ of s for (local) evaluation. Suppose that another task T' needs to evaluate some s_i. In this case, observe that T' need wait only until graph s_i is evaluated. However, task T will complete only after evaluating all of $s_1, ..., s_k, s$. Thus, if T' waits on T instead of s_i, we forego parallel execution of T and T'. Also, note that T selectively waits for one or more subexpressions, and is not restricted to await *all* subexpressions it has created before it can resume from any one of them. That restriction is imposed in [4], and requires measures to avoid deadlock.

5 Queue Management

In EQUALS, new or awakened tasks are placed in a global ready queue from which they are taken by free evaluators[5]. Since every task suspension and resumption involves the queue, it must be carefully designed so it is not a system bottleneck. However, a global queue enables a simple mechanism for load sharing. We describe a low-overhead scheme to implement the global queue, and a mechanism for its use to estimate, balance, and share the workload.

5.1 Ready Queue

For the ready queue, consider using a linked list of tasks. With this implementation, the queue must be locked to add or remove entries; hence, the ready-queue lock can become a system bottleneck. Furthermore, a heavy burst of bus traffic is generated when all idle evaluators race to retrieve a newly enqueued item.

Alternately, the queue can be implemented as a sequence of *slots*, each holding at most one task. As each idle evaluator arrives at the queue, it is assigned the next available slot. If that slot holds a task, the evaluator picks up that task for execution; otherwise, the evaluator polls its empty slot until it is filled with a task. Note that when other slots are filled, this evaluator is oblivious, and the evaluators do not race one another. For tasks, one arriving at the queue is placed in the next empty slot. Thus the access to the queue is serialized using two independent locks — a tail lock for producers and a head lock for consumers. This *split-lock* implementation of the queue resembles the dual queue used by [2].

Note that the split-lock queue can be easily implemented with a circular array of task pointers. Furthermore, the head and tail can be advanced with (atomic) locked-increment instructions. However, in such an implementation, there

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

is a possibility of *queue overflow*, since the maximum queue length is fixed (2^{16} entries in our implementation). To manage this situation, we keep a separate overflow queue. Idle evaluators are used to move tasks from the overflow list into the ready queue.

Measurements show that access to the split-lock queue has a low overhead and the queue is not a system bottleneck. For instance, enqueuing requires 7μs and dequeuing requires 5μs; in the worst case, the time requirement for either operation grows less than linearly with the number of processors using the queue. In contrast, the simple linked-list implementation has same base enqueue and dequeue times, but its behavior degrades rapidly with the number of processors, even in practice. For example, the *Tak* program stresses the queue, and the split-lock implementation began with enqueue and dequeue times of 9μs and 5μs, respectively, on 2 processors. From there, times increased to 38μs and 22μs on 10 processors. For the same program, the enqueue times for the linked-list implementation increased from 9μs on 2 processors to 131μs on 10 processors[6].

Note that enqueue and dequeue times directly affect system performance. For instance, with the linked-list implementation *Tak* has speedup of only 5.0 on 10 processors, versus 8.5 with the split-lock implementation.

5.2 Load Control

For many parallel programs, the available parallelism may far exceed the available parallel resources. In such conditions, tasks would be executed sequentially by the available processors, and the system's load would be high. An important optimization avoids creating tasks under such conditions. In EQUALS, when the number of tasks in the ready queue exceeds a threshold, the evaluators avoid creating tasks and instead perform the intended computation locally. This technique has been used effectively in previous implementations [6], and is implemented with hysteresis in EQUALS as follows.

The EQUALS system may be in parallel (low-load) or sequential (high-load) mode, and this information is maintained by a flag. Accesses to this flag are highly optimized: the flag is updated only when a task is enqueued or dequeued, and it is read or written without any locking, since it is advisory. Furthermore, most accesses are to read this flag and can be satisfied by the local cache; thus, this flag is not a system bottleneck.

To switch between modes, the system compares the number of tasks in the ready queue, N, against two thresholds N_{seq} and N_{par}, as follows: In parallel mode, when N exceeds N_{seq}, the system enters sequential mode; in sequential mode, when N falls below N_{par}, the system switches to parallel mode. The values of N_{par} and N_{seq} have been determined through experiments. Though there is no ideal setting for all programs, $N_{\text{par}} = 1$ and $N_{\text{seq}} = 20$ yields good overall results.

6 Performance

We next present the performance of EQUALS on example programs, to illustrate the effectiveness of the techniques discussed in this paper. The examples chosen are: *Euler*, which computes the Euler totient function from 1 to 1000; *Queens*, which finds all solutions to the n-queens problem on a 10×10 board; *MatMult*, which multiplies two 100×100 matrices; and *Tak*, the computation of the Takeuchi function $tak(21, 14, 6)$. Of these programs, the first two are memory intensive, creating and destroying many list structures. *Euler* and *MatMult* have large grain sizes. *Tak* is compute intensive, with little heap manipulation, but creates a large number of fine-grained tasks.

These examples have been selected to test the performance of various parts of the runtime system. First, we present information on the heap, stack, and task usage for each example program, in table 1. In that table, the columns 'Tot. Alloc' list the total number of allocations for each of the objects and 'Max. Alloc' gives the maximum number of objects simultaneously held by the system. The number of nodes allocated varies very little when number of processors changes, and hence is given for one processor only. From the table, we see that *Euler* allocates a very large number of nodes, and hence stresses the heap management system, but creates few tasks. *Queens*, on the other hand, creates many tasks *and* allocates many heap nodes. *Queens* and *MatMult* show the importance of handling shared structures effectively, since both generate many shared structures (indicated by the ratio of dereferences to allocations). In *Tak*, the number of tasks and stacks allocated are the same, indicating that no task exceeds the initial stack allocated to it. It should be noted that *Tak* creates half as many tasks as *Queens*, even though

[6] Dequeue times are not given since we cannot separate dequeue times from idle time in the linked list implementation.

Program	Num Procs	Nodes Tot Alloc	Nodes Max Use	Nodes Derefs	Stack Tot Alloc	Stack Max Use	Tasks Tot Alloc	Tasks Max Use
Euler	1	1,311,206	3,001	2,116,866	58,612	30	1	1
	10				57,221	1,105	1,001	1,001
Queens	1	524,409	65,273	5,699,262	180,430	128	1	1
	10				49,615	168	21,522	8,706
MatMult	1	40,204	40,102	4,000,203	16,228	4	1	1
	10				16,577	591	2,265	673
Tak	1	2	1	2	1	1	1	1
	10				11,076	81	11,076	145

Table 1: Memory usage Statistics

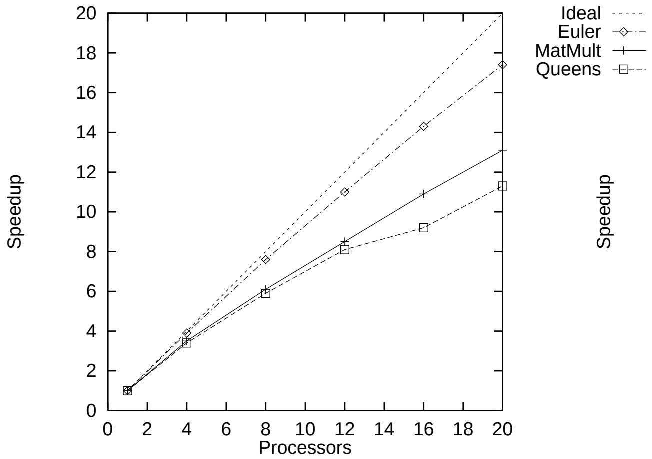

Figure 4: Speedup of example programs on EQUALS.

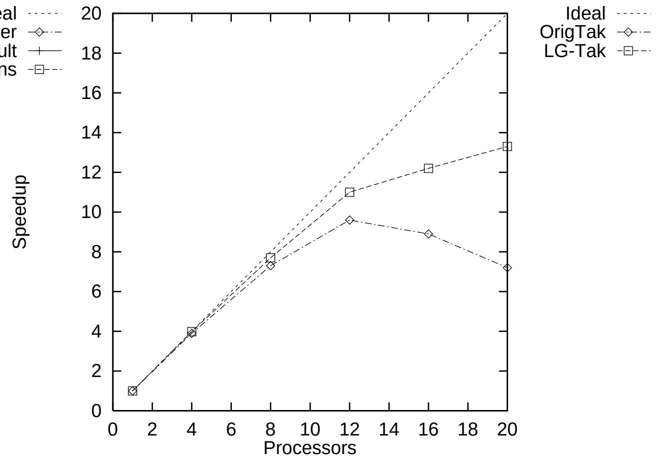

Figure 5: Effect of grain-size control on *Tak*.

Tak takes only one-tenth of the execution time as *Queens*. The large number of tasks is a symptom of the small grain sizes in *Tak*, which stresses the task and queue managers.

Speedup Curves Speedup curves for *Euler*, *Matmult*, and *Queens* appear in figure 4, and demonstrate speedups comparable to those in [1]. We note that absolute times are also comparable, and that our results account for all costs of memory management, whereas the timings in [1] omit garbage collection. The speedups achieved by *Euler*, which generates intense heap activity, indicates that the memory manager is highly scalable. Next, we examine the speedups achieved on *Tak*, which scales well until 12 processors, but then degrades. Too many fine-grained tasks are apparently being created, and the overhead of their creation and synchronization has become significant. Often, this overhead even exceeds the amount of useful work the task must do. *In fact, over 65% of tasks created in this example have computation smaller than 150μs, the parallel overhead.* Furthermore, the number of such tasks increases from $2k$ on 2 processors to $12k$ and $26k$ on 10 and 20 processors respectively.

By avoiding creation of parallel tasks for function calls that directly compute the base case, the program scales to 20 processors, and only 25% of tasks are smaller than the parallel overheads. Also, there are only 750 such tasks on 2 processors and $3k$ each on 10 and 20 processors. The modified and original programs, denoted *LG-Tak* and *OrigTak*, respectively, are compared in figure 5. Of course, such grain-size tuning is not always possible, and parallel overheads can eventually overwhelm and degrade the performance of some programs.

7 Conclusions

In this paper, we described the design and implementation of a runtime system for EQUALS, consisting of a scalable memory manager and a simple and efficient task manager. We demonstrated the effectiveness and potential impact of our design on the parallel performance of programs.

While performance is generally very good, future improvements in EQUALS remain possible, and we are actively improving the system. Runtime enhancements are only one part of planned and ongoing work. We are investigating load-estimation strategies and their use in preventing creation of fine-grained tasks. Efficient support for stream parallelism is an important area of study, since many programs with insufficient (horizontal) parallelism exhibit stream parallelism. In summary, there is much interesting work ahead.

Acknowledgements We thank I.V. Ramakrishnan for his guidance and encouragement. Use of a Sequent Symmetry S81 was provided by the Department of Computer Science at Rice University under NSF Grant CDA-8619393, and another Symmetry was provided 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

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

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

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

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

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

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

[7] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 1992.

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

[9] S. Wray and J. Fairbairn, Non-strict languages—Programming and implementation. *The Computer Journal*, 1989.