

A Novel Approach to Parenting in Functional Program Evaluation

Julian R. Dermoudy

School of Computing
University of Tasmania
GPO Box 252-100, Hobart 7001, Tasmania

Julian.Dermoudy@utas.edu.au

Abstract

The ability for multiple threads to enter the same graph node without contention and conflict is a necessary component of the graph reduction of functional languages since graph components may be shared. Shared closures, however, compound the difficulty of priority management.

The original GUM runtime system does not track which threads require the evaluation of which closures or which sparks relate to which threads. These problems are remedied in the novel implementation of GUM presented here which introduces dynamic thread hierarchies and supports prioritised scheduling.

Keywords: Concurrency, Distributed Systems, Functional Programming.

1 Introduction

1.1 Concurrency in Functional Languages

Functional programming languages and compilers have received much attention in the last decade for the contributions possible in parallel execution. Since the semantics of languages from the functional programming paradigm manifest the Church-Rosser property (that the order of evaluation of sub-expressions does not affect the result), sub-expressions may be executed in parallel [Barandregt 1984]. The absence of side-effects and the lack of state facilitate the availability of expressions suitable for concurrent evaluation.

There are two schemes for evaluating a functional program concurrently: *conservative evaluation* and *speculative evaluation* [Peyton Jones 1989]. Under the former, code fragments are only executed when it is certain that the value of the expression is required, *i.e.* *lazy* semantics hold. An example of the use of concurrent evaluation under the conservative evaluation scheme is given in Figure 1.

```
f :: a -> a -> a -> a
f x y z = if (x < y) then y else z
```

Figure 1: An example function.

Under a speculative evaluation scheme, laziness is usurped by eager concurrent evaluation of expressions

when there is some likelihood that the values of those expressions will be needed. In the example of Figure 1, all three arguments could be evaluated concurrently if sufficient computing resources were available.

Annotations are meaning-preserving program decorations [Peyton Jones 1989]. They do not affect the outcome of the program in the sense that ignoring them will result in sequential program execution.

Annotations can be used to indicate to the compiler where sufficient parallelism may be found, and when to avoid the penalty of communication on threads of small granularity. Annotations are also used to assist the scheduler to select between threads; this is done by providing a thread ‘priority’. If a thread is known to contribute to the outcome of the program it is said to be *mandatory*. If a thread is known not to contribute to the outcome of the program, it is said to be *irrelevant*. A thread is said to be *speculative* if its contribution to the program’s outcome is not yet known.

1.2 Evaluation

Many functional language evaluators (including the GHC compiler [Hall, Hammond, Partain, Peyton Jones, and Wadler 1992]) conceptually represent the compiled functional program as a graph in which functions/values are the vertices and edges represent the relationship of function application and connect a function with its arguments. As evaluation of the graph proceeds, the graph is rewritten by replacing those parts of the graph that have been computed with their result — thus simplifying the graph. The graph is progressively reduced due to the rewriting; the process is called *graph reduction* [Peyton Jones 1987].

Within an implementation, a node in the graph is represented as a *closure* which contains the values conceptually represented in the graph node plus other implementation-facilitating information, such as a type field (*tag*), and various other fields depending upon the value of the tag. Examples include: a literal value, an argument count, and a list of pointers to (addresses of) parameters [Peyton Jones 1987].

Evaluation of the user’s program utilising graph reduction occurs by mapping threads that refer to closures to tasks that perform the evaluation. This mapping is dependent upon a number of runtime policies (elaborated upon in [Dermoudy 2002]) but ultimately is dictated by the hardware architecture.

1.3 Impediments to Parallelism

Under conservative evaluation, only those threads known to contribute to the final program outcome (*mandatory* threads) are considered for execution. The amount of parallel reduction is thus dictated by the problem being solved (although it is bounded by the greater of the concurrency present and the number of processing elements available).

Conservative evaluation comprises an allocation of threads to processing elements on demand. It is conceptually simple and is the approach usually adopted by runtime systems. There are no unnecessary overheads, the thread of execution is deterministic, there is no competition between threads of differing priorities for resources, and runtime errors should manifest when encountered.

Conservative evaluation can, however, delay the execution of program fragments which, although able to be evaluated earlier, are not evaluated until it is determined that their computation is required. Since the aim of concurrent evaluation is to reduce the wall-clock time for program execution, the concurrent execution of these program fragments should be attempted whenever it is possible (and beneficial) to do so.

Under speculative evaluation [Peyton Jones 1987] idle¹ tasks speculate on the future relevance of threads that are not as yet known to be crucial to the outcome of the program. If subsequently the value is required, it may have already been calculated.

Priorities could be assigned so that (for example) mandatory threads have a higher priority than speculative threads spawned by a mandatory thread. These in turn could have a higher priority than speculative threads spawned by a speculative thread, and so on. Lastly, those threads suspected to be irrelevant could have the lowest priority assignable so that they are never scheduled.

A runtime difficulty of speculative evaluation is the fact that the priority of a speculative thread can change. Consider the example:

```
if p then x else y
```

Evaluation of x and y could begin concurrently with the evaluation of the predicate p . Once the outcome of p is deduced the priority of the threads evaluating x and y must be changed. If p reduces to `True`, then the thread evaluating x must become mandatory whilst, conversely, the thread evaluating y must become irrelevant. If x and/or y consist of sub-threads, then the priorities of these must also change. The possibility also exists that these threads will be under evaluation — perhaps on remote processing elements.

The difficulty of priority management is compounded in the functional programming context by sharing. Since no side-effects exist, it is possible, and indeed, advantageous

for an expression to be shared by a number of other functions/expressions within the program. Consider the following contrived example:

```
power(x,n) =  
  if n==1 then x else power(x*x,n-1)
```

Evaluation of the expression x should only occur once, even though through speculative evaluation, it could be requested three times (or four times if the lazy semantics of the evaluation of function arguments is abandoned). Hypothetically, if there is spare capacity for only one speculative thread in addition to the mandatory evaluation of the comparison of n with 1, the consequent, x , could be selected. Now it is quite possible that the majority of the evaluation of x could be complete at the point the comparison returns `False`. It would be wasteful to revoke the evaluation of x since it is plain that this value will still be required in the contra-consequent. The priority of the thread evaluating the consequent, however, will need to be downgraded to irrelevant, and execution of the contra-consequent will need to commence at mandatory priority. Thus the evaluation of x will have two (three) priorities simultaneously: irrelevant for the consequent thread, and mandatory (twice) for the contra-consequent thread.

Therefore, not only must the priority management system be flexible enough to stall a thread without discarding the work done by it, the system must also be capable of allocating multiple priorities to a thread and selecting the most important of these at all times.

To cope with these issues the following strategies have been implemented by the author:

- threads may be assigned speculative priorities, with the extremes of this priority range *irrelevant* and *mandatory* — a priority-based scheduler will ensure the evaluation of the most important threads at all times; and
- a thread's speculative priority should be variable — it should be possible to upgrade and downgrade a thread's priority to reflect new information regarding the relationship between that thread's result and the program outcome.

1.4 Platform

1.4.1 Language

GPH is an abbreviation for Glasgow Parallel Haskell — and is often written G_pH to align it with Parallel Haskell [Aditja, Arvind, Augustsson, Maessen, and Nikhil 1995], which is denoted pH . GPH is a super-set of Haskell and extends Haskell with two primitive functions `par` and `seq` [Trinder, Barry, Davis, Hammond, Junaidu, Klusik, Loidl, and Peyton Jones 1998].

The GPH expression `p `par` e` has the same value as expression e and is not strict in its first argument. It is used to indicate potential parallel evaluation. The expression p may be evaluated by a newly created thread while the original thread continues its execution of expression e . Expression p is said to be a *spark*. If a thread to evaluate p is created, the act of converting the

¹ “Idle” is a relative term; a task is “idle” if it is executing no threads, or is executing a thread of lesser importance than a thread waiting for execution by another task, or on another processing element.

spark into a thread is called *sparkling*, and p is said to have been *sparkled*.

1.4.2 Operation

The parallel evaluation system of the publicly available version of GHC — and in particular its speculation and load distribution facilities — are somewhat rudimentary. The runtime system includes only one annotation for parallel evaluation, and once an annotated expression is sparked into a thread it competes equally with mandatory threads and other ‘speculative’ threads as each task executes all runnable threads in a round-robin fashion until its thread pool is empty.

At runtime in parallel executions, the compiled user’s program is combined with the “Graph reduction for a Unified Machine model” (GUM) runtime system (see [Trinder, Hammond, Mattson Jr., Partridge, and Peyton Jones 1996; Hammond, Loidl, Mattson Jr., and Partridge unpub; Trinder, Barry *et al.* 1998]) and the result executed.

Although GUM had its inception on the specialised GRIP architecture, it is now architecture neutral and portable [Trinder *et al.* 1996; Trinder, Barry *et al.* 1998] using a globalised address space. (Each processing element has a local heap, which is independently garbage collected; the collection of all local heaps provides a virtual global heap.) Execution relies upon communication between virtual processing elements facilitated by the use of a hardware abstraction — PVM [Geist, Beguelin, Dongarra, Jiang, Manchek, and Sunderam 1994].

GUM comprises one System Manager task with the remaining processing elements being ‘workers’. Each worker task has a copy of both the program and the runtime system with one processing element known to contain the ‘main program’ thread. At start-up, the system manager spawns the worker tasks and synchronises them. Each task then executes the runtime system which results in program evaluation, load distribution, garbage collection, *et cetera*. When the main thread terminates, the task indicates this by communicating with the System Manager which then initiates and coordinates the shutting down of all tasks.

1.4.3 Characteristics

Threads may be in any one of five states during execution [King, Hall, and Trinder 1998]:

- running;
- runnable (waiting to be scheduled);
- blocked (waiting for another thread to complete);
- fetching (waiting for a value to arrive from a remote processing element); or
- migrating (moving a thread from a busy processing element to an idle processing element).

A pool of runnable threads is maintained on each processing element and each thread is executed for a user-controllable time quantum before context switching occurs. All threads have the same priority and so the thread pool is implemented as a simple queue.

The runnable thread pool is implemented as a linked list referenced by two pointer variables: a *head* and a *tail*. Each node in the list represents the state of a thread when not executing and is called a *thread state object* (TSO). One of the fields in a TSO is a *link* field which is used to reference the next TSO in the list. This field is manipulated directly when a TSO is removed from, or placed into, a list of threads.

A major use of the ‘link’ field is in the implementation of *blocking queues*. A blocking queue is a list that contains TSOs waiting for some event. This event may be the evaluation of a closure, input/output, the arrival of a value from a remote processing element, *et cetera*. When blocking queues are implemented, the TSO for a runnable thread waiting for an event is removed from the runnable thread pool and placed in an appropriate blocking queue. It does not return to the runnable thread pool until the event has occurred.

Closures in the graph manipulated by the GUM runtime system all have a common structure as shown in Figure 2 (adapted from [Loidl and Hammond 1994] and [AQUA 1996]).

| Header | | Pointers | Values |
|--------------|-----------------|---------------|-------------------|
| Fixed Header | Variable Header | Pointer Words | Non-pointer Words |

Figure 2: Closure structure.

Closures are of specific types; TSOs have the closure type TSO. This is one of a collection of so-called “specialised” (SPEC) closure types [AQUA 1996].

2 Problems

2.1 Anonymity

When a spark is created, the sparked expression occurs textually within the current thread of execution. This relationship is lost in the unmodified GUM implementation when sparking occurs. It is not lost in the augmented implementation presented here. Instead, the identity of the thread that gives rise to the creation of the spark is stored in the spark. This thread identity is drawn from the TSO. When the spark pool entry is sparked, this information is passed to the newly created thread and is available for profiling and/or debugging analysis.

2.2 Unknown Parents

In the original GUM system, the first time a closure is entered by a thread, that closure is converted into a *black hole* and the thread continues to evaluate the closure. No record is made in the closure of which thread is evaluating it. When subsequent threads enter the closure their TSOs are placed on the closure’s blocking queue.

Since the closure is shared by all of these threads and the first thread is evaluating the closure under the evaluate-and-die evaluation model [Peyton Jones, Clack, Salkild, and Hardie 1987; Loidl 1998], all of these threads are in fact related: each thread on the blocking queue is a *parent* of the thread initially entering the closure, and it is a *child*

of theirs. Ignoring such relationships will result in wasted processing element and erroneous scheduling decisions — both of which reduce effectiveness.

Consider the expression:

```
if p then (if q then x else y) else y
```

Using annotations, this expression may be re-written as a function, *m* say, as shown in Figure 3.

```
m p q x y = par r (par y s)
           where r = par x (par y t)
                 s = if p then r
                     else y
                 t = if q then x
                     else y
```

Figure 3: Example functional code.

Assuming that the evaluation of *m* is required and that branches of conditionals are equally likely to be executed, the value of *p* is required. If processing resources are available it could be speculated that the consequent, *r*, (itself a conditional) and the contra-consequent, *y*, will be needed and these could be executed at a reduced priority (50% each say). The consequent will require the execution of *q* (at 100% of 50%) as well as the speculative evaluation of *x* and *y* (both at 50% of 50%). In a functional programming environment, the evaluation of *y* will only occur once. Note, though, that it is required by a thread of priority 50% that is the contra-consequent of the outer conditional, as well as by a thread of priority 25% that is the contra-consequent of the inner conditional. Therefore, the highest priority (50%) should be assigned to it. The lesser priority, however, cannot simply be discarded — if the predicate *p* evaluates to the Boolean value `True`, the contra-consequent is now irrelevant which would give *y* a priority of 0%. The predicate *q* should now be upgraded to mandatory which has the consequence of necessitating an increase in the priority of both *x* and *y*. The priority of *x* requires an upgrade to be 50% of 100% while the priority of *y* should also be altered to 50% of 100% and not 50% of 0%. The hierarchy of the potential threads and their priorities is shown in Figure 4.

For effective speculative evaluation, each closure, therefore, should retain information about which thread is evaluating it, and each thread must retain a vector of priorities (in descending order for efficiency) for *all* containing threads. Given that the priorities of the containing (parent) threads may change, the priority of the contained (child) threads may hence change also.

2.3 Thread Death

Another issue is the death of threads. When a mandatory/speculative thread completes the required evaluation, the thread dies. At this point the thread's children are no longer required by this thread, and could be terminated (as could the child threads of those child threads, and so on). There are four options:

- do nothing to those child threads, and let their priority and existence remain;
- perform no priority change but terminate the child threads;

- change the priority of those child threads to irrelevant and then terminate each one; or
- change the priority of the child threads to irrelevant but don't terminate them.

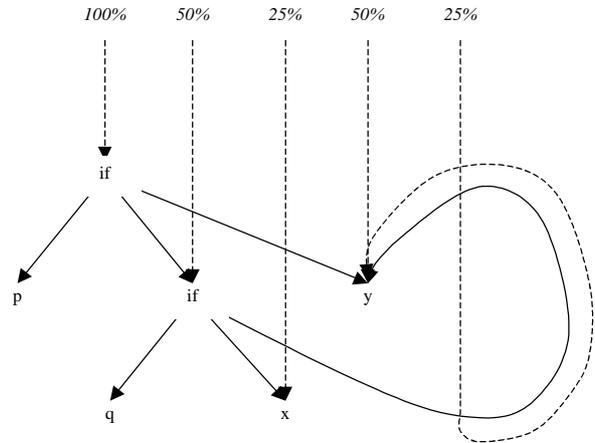


Figure 4: Thread hierarchy.

Not altering the status of child threads at all will potentially perpetuate the execution of now unnecessary threads at the expense of executing threads that may be needed more. If the priority of the child threads isn't changed but the child threads are terminated (and so on for those threads' children) the false competition with other potentially needed threads will not occur — which is an improvement. Due to sharing, however, a closure being evaluated by a child thread may subsequently be required by an alternative parent and hence terminating the child thread and cancelling the evaluation would be a waste of effort as the closure will require re-evaluation. If the priority of such threads is modified and then the threads are terminated — as dictated by the third option — the process will involve two steps rather than one, which would potentially allow extra execution of now unnecessary threads, although this would be short-lived. This scheme will still involve potentially premature termination of required evaluation. The final option involves changing the priority of the child threads (and their child threads, *et cetera*) to irrelevant but not terminating the threads. At worst this option will leave irrelevant threads in the runnable thread pool which will occupy storage. These threads will never be scheduled for execution unless idle tasks exist, and therefore they are not detracting from the execution of mandatory threads — unless and until memory is exhausted. The final option has been implemented; see [Dermouly 2002] for more information.

3 Solutions

3.1 Spark Parent Identification

A TSO possesses an id that uniquely distinguishes an arbitrary TSO from all other TSOs on the same processing element. TSO ids are not unique across the multicomputer, however, and cannot solely identify a particular TSO from all others. In the case of sparks that may move from one processing element to another due to

load distribution [Dermoudy 2002] (or processing element allocation annotations [Dermoudy 2002 and 1999]) such an identifier is inappropriate and insufficient. A global identifier is required that will be unique for all TSOs. TSOs can be uniquely identified by a global address since:

- global addresses are already used for uniquely identifying closures across the multicomputer;
- the local address of a closure with a global address is readily available *via* an indirection table; and
- a TSO is simply a type of closure.

In the extended implementation of GUM described here, a spark is associated with the global address of its parent thread as follows: when a spark is created, the TSO for the current thread (the spark's original parent) is *globalised* (allocated a global address) and the resulting global address is stored in the spark. Three new fields are required: one for the global address of the parent thread's TSO, one to act as the head of a list of parent threads, and one to act as the head of a list of child threads.

When a spark is converted into a thread, this global address is used to initialise the parent list. The child thread (formerly the spark) thus knows of the existence and identity of the parent, but the parent does not know of the existence of the child. The remedy to this problem is presented in Section 3.3.2.

3.2 Thread Parent Identification

The first change that is required to the original GUM runtime system to allow the management of thread relationships is the inclusion of a global address in all closures. There must be some record of which thread is evaluating a closure so that subsequent threads may become parents of this thread.

Since threads may not be moved from the processing element on which they are created, the record could be achieved by either including a global address or TSO id in the closure, or by including a list of closures that have been — or are being — evaluated by a thread in the TSO. Since the evaluate-and-die model of threads is used, the same thread will evaluate possibly many closures and each closure must be associated with the thread — hence the list. The overhead of this is high: first each TSO must be considered, and secondly each TSO's list must be searched. To complicate matters further, TSOs are not all stored in the same place; some will be in the runnable thread pool, others will be on various blocking lists.

The use of a TSO id is also an alternative, but TSO ids are not globally unique (as stated in the previous section), and hence the use of a TSO id is not compatible with future implementations involving thread migration/allocation [Dermoudy 2002]. Locating a local TSO on a processing element given only its TSO id is also a problem as no unified collection of TSOs exists.

The use of a global address again solves these problems. There is the question of how many closures need to be exposed to a global address: all, or those unevaluated closures whose evaluation is required by multiple threads. The minimum list of closure types is those that

potentially containing blocking queues. For simplicity and orthogonality, space for a global address has been added to all closure types.

When a closure is entered by a thread, the thread's TSO closure is globalised if this has not already occurred, and the global address for the TSO is stored in the parallel information field within the fixed header of the root closure. The overhead of this is small since each TSO requires only a single global address and threads are relatively long-lived since the evaluate-and-die model is utilised.

3.3 Thread Priority Calculation and Adjustment

3.3.1 Preamble

There are three cases where a thread's priority will change:

- at its creation;
- when the priority of one of its parents changes; and
- when it dies.

Each of these will now be discussed.

3.3.2 Sparking

The first case is the creation of the thread. Recall that a spark possesses fields that hold its annotated priority and its parent (containing) thread TSO's global address. The priority indicated in the program is a relative priority and depends upon the prioritised context of its parent; the annotation provides a priority *factor* that must be multiplied by the current priority of the parent thread. Therefore, when a spark is converted into a thread, its parent list can be initialised with the values stored in the spark: the parent thread's TSO's global address and the priority factor. In order to obtain the full contextual priority, the parent must be contacted. The existence of global addresses can help here. If the parent thread is local, its priority may be readily extracted and stored in the new thread's parent list entry. The new thread's priority can then be calculated and the thread can be added to the runnable thread pool. If the parent thread is remote, a message is sent to the thread requesting its current priority, the new thread's relative priority temporarily becomes an absolute priority, and the new thread is added to the runnable thread pool². If the parent thread has terminated, the relative priority also becomes the absolute priority. (Thread termination may be detected by checking whether the global address is associated with the current processing element or is a *null* local address.) Figure 5 illustrates the process.

When a newly created thread contacts its parent thread it advises the thread of its existence by providing it with its newly acquired global address (the TSOs of new threads are globalised upon creation). The parent responds by

² In a multicomputer the process of contacting a remote parent will incur a latency — the relative priority is temporarily used to mask this.

adding the child thread's TSO's global address to its child list and advising the child thread of the parent thread's current priority. Unlike the parent list, only the TSO's global address is stored in the child list.

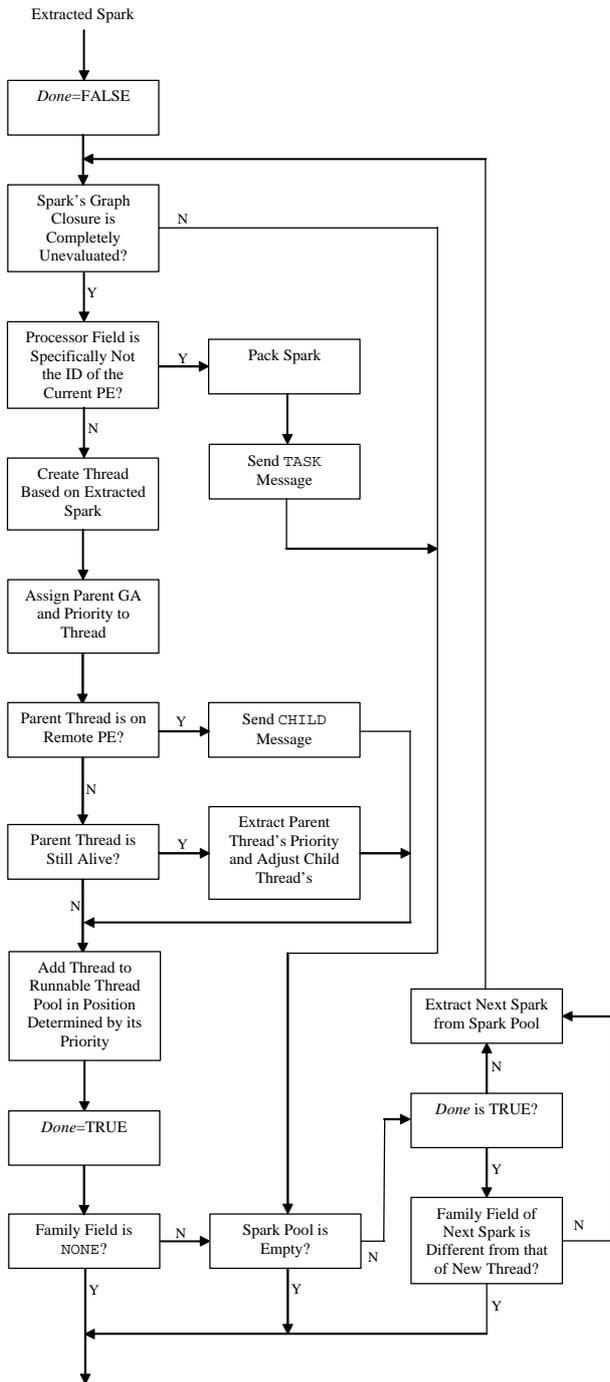


Figure 5: The detailed process of sparking (including parent and priority identification).

3.3.3 Changing a Thread's Priority

When a thread's priority changes, the change is advised to all of that thread's child threads. This is the purpose of maintaining the child list. If the child is local this may be done directly, if the child is remote a message must be sent. The latter scenario is discussed below. Each thread stores its current priority as well as that of its parent threads. If the priority of a parent thread is changed, then

the entry in the parent list is updated to reflect this and the new priority is compared with the thread's current priority. If the new priority is higher than the current priority the current priority is upgraded. Similarly, if the old priority was the thread's current priority and this is decreased, then the parent list is searched for the highest priority value and this becomes the thread's current priority — thus the priority is downgraded.

3.3.4 Thread Termination

The third case is thread termination. When a thread terminates it advises its parent and child threads of its demise (through the use of a message — see Section 3.5.6 — if a parent/child thread is remote). It also replaces its local address with a *null* value. The parent threads react by removing the child thread's TSO from the list of child thread TSOs; the child threads react by interpreting the news as a change in priority of the parent thread to irrelevant, processing this (as described in the preceding section), and then removing the parent thread's TSO's global address from the parent list. All such changes are processed when the next reschedule is performed; all changes are made to thread TSOs and hence suspended threads do not need to be re-awoken.

3.4 Dynamic Thread Hierarchy Management

A thread entering a 'black holed' closure must register as part of the thread hierarchy with the original thread. If the second/subsequent thread is speculative, it may have been moved from its original processing element by load distribution. In this case, a specialised message is sent to the first TSO advising it of the existence of the new parent. The first TSO's identity can be obtained by extracting the global address from the closure. If the first TSO is local no message is sent but the advice is issued.

The advice consists of the new parent thread's TSO's global address (if the TSO has not been globalised then this is done prior to contacting the original parent thread) and the parent thread's current priority. These pieces of information are stored in a new entry in the original thread's TSO's parent list (the priority factor is set to be 100% for newly found threads).

The new thread adds the global address of the original thread's TSO to its child list by creating a new element³.

The general process is illustrated in Figure 6.

Eden (see [Dermoudy 2002; Klusik, Peña, and Segura 2000]) also contains an analysis of thread hierarchies and communication that detects and removes chains of indirections. An Eden thread must communicate with its parent when results are returned and this may require communication between processing elements. The parent-child hierarchy identified here is not for result reception but for priority management. If a parent thread changes its priority it informs its child threads. If the priority of one of these child threads changes, then that

³ Mutual dependency is catered for — a TSO can only be added to a parent list once and to a child list once, and cannot be added to a child list if it is already known as a parent.

thread's children, in turn, are advised of the change in priority. Chains of indirections can thus occur here also. It should be noted, however, that this is necessary as a change in priority should be advised to all threads in the hierarchy that the change affects; bypassing would not be appropriate.

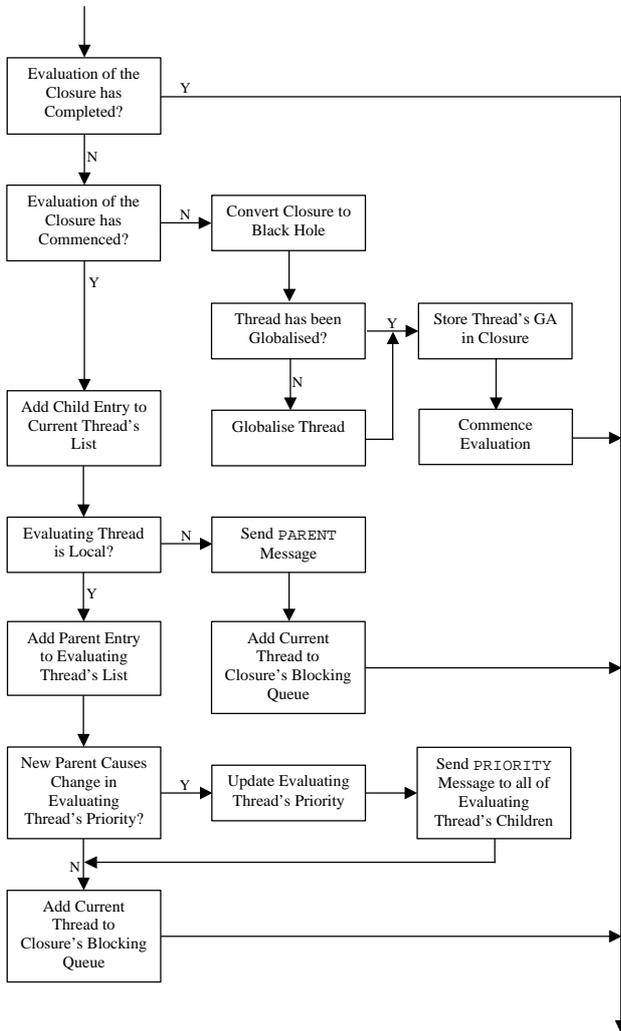


Figure 6: Closure entry behaviour.

3.5 Messages

3.5.1 Preamble

Six new messages have been added to the GUM runtime system to support dynamic thread hierarchy management. Each will now be presented in turn.

3.5.2 The CHILD Message

When a spark is converted into a thread the new thread must calculate its priority and inform its parent thread of its existence. If the parent thread is on a remote processing element this process must be performed through the use of a message. The message CHILD is used for this purpose and communicates to the remote TSO the new thread's TSO's global address. Processing the CHILD message is performed by the GUM runtime

system which updates the appropriate fields of the parent thread's TSO.

The GUM runtime system replies to the initiating task with the current priority of the parent thread contained within a PRIORITY message. If the parent thread has terminated, the priority value irrelevant is returned. Along with the priority, the sending (parent) thread's TSO's global address and the recipient (child) thread's TSO's global address are communicated.

3.5.3 The PRIORITY Message

The PRIORITY message is used to communicate a new or changed thread priority to a child thread. The values communicated are the destination (child) thread's TSO's global address, the sending (parent) thread's TSO's global address, and the sending thread's priority.

If the reception of the PRIORITY message alters the child thread's priority then PRIORITY messages containing this new priority are sent to all the child thread's child threads. If not, then there is no response to the PRIORITY message.

If the destination global address maps to null, the child thread has since terminated. In this case the PRIORITY message is simply discarded.

3.5.4 The PARENT Message

When a thread enters a closure that is already under evaluation by another thread, the former thread becomes a parent of the latter. To indicate this, the newcomer sends a PARENT message to the original thread providing its TSO's global address and current priority. This message is processed by the GUM runtime system as described above.

If the reception of the PARENT message alters the child thread's priority then PRIORITY messages containing this new priority are sent to all the child thread's child threads. If not, then there is no response to the PARENT message.

3.5.5 The THREAD_GA Message

When a thread enters a closure held on a remote processing element, a FETCH message ensues. In the context of dynamic thread hierarchies, this thread is a parent of the thread evaluating the remote closure. To record this, a THREAD_GA message is sent by the task receiving the FETCH message. The THREAD_GA message contains the global address of the thread's TSO for the thread that is evaluating the closure locally.

Upon reception of the THREAD_GA message, the enclosed child thread's TSO's global address is stored in the FETCHME closure (just as it is in the remote closure) and the FETCHME closure's blocking queue is examined. For each thread blocked on the FETCHME closure (there will be none initially, but when the FETCHME closure has been converted to an FMBQ closure type the list will be non-empty and the same code is used), its TSO's parent list is examined to ensure that the child thread's TSO is not

present and if not, the child thread's TSO is added to the parent thread's TSO's child list. The parent thread's TSO's global address is then returned to the child thread through the transmission of a `PARENT` message. This process is repeated whenever a new thread enters the `FETCHME/FMBQ` closure.

3.5.6 The `ZOMBIE` and `TSO_DEATH` Messages

When the reduction activity of a thread is completed, the thread's work is also complete and the evaluated closures can be rewritten. This completion obviates the parent-child relationship involving the (child) thread that has now finished evaluating the closure that is being rewritten and the (parent) threads that have blocked on the closure waiting for its reduction to occur. To indicate that the reduction has completed and the sharing of the closure is concluded a message is sent to each child and parent thread of the terminating thread: the `ZOMBIE` message. This message incorporates both the sending thread's TSO's global address and the receiving thread's TSO's global address.

When the `ZOMBIE` message is received, all child threads are sent `PRIORITY` irrelevant messages by the recipient task.

After the `ZOMBIE` message has been sent, the business of rewriting the closure and thread termination is then undertaken by the terminating thread. When the thread that has re-written the closure actually terminates, a `TSO_DEATH` message is sent with the sending thread's TSO's global address, the receiving thread's TSO's global address, and whether the terminating thread is a child or parent of the intended recipient.

When the `TSO_DEATH` message is received the following occurs:

- if the terminating thread is indicated as a child of the recipient then the child list is examined. If the sending thread's TSO's global address is found then the entry is removed from the child list. No reply is forthcoming.
- if the terminating thread is indicated as a parent, then the sending thread's TSO's global address is searched for in the parent list. If it is found then the entry is removed from the list and the recipient thread's priority is re-calculated. If the priority changes then all child threads are advised of the change through the sending of `PRIORITY` messages.

4 Implementation

4.1 Preamble

To successfully implement the extensions described here a number of data structures needed to be added while others needed extension and/or modification.

4.2 TSOs

The TSO structure required modification to allow the storage of a priority and list of child and parent TSOs.

The list of child TSOs comprised a linked list of nodes with a global address and connecting pointer, while the parent TSOs contained the global address of the parent TSO, the last known priority of that TSO, the relative priority (factor) of the annotation, and the link.

4.3 Closures

To facilitate the dynamic management of thread hierarchies and accurately associate the correct priority with a closure evaluation, the relationship between a thread and the closure it was evaluating needed to be explicitly recorded as this was not done in the distribution software. To connect the closure with the thread evaluating it (which may potentially be on any processing element), the fixed header of each closure was enlarged by one field to allow the storage of a global address. Each thread was then globalised and its global address stored in the global address slot of the closure's fixed header.

4.4 Messages

In the distributed implementation of dynamic thread hierarchy management, it is quite possible for sparks to have been transferred to alternative processing elements for execution and thus be remote to the thread that spawned the spark. Therefore, to manage thread hierarchies which involve parent and child thread identities and priorities, as well as thread termination, six management-related messages were necessary. As discussed, these messages are `CHILD`, `PARENT`, `PRIORITY`, `THREAD_GA`, `ZOMBIE`, and `TSO_DEATH`.

4.5 Dynamic Thread Hierarchy Management

The entry and update code for closures was altered for a number of closure types. The entry code was altered to include globalisation of the TSO and the storage of the TSO's global address in the new field of the closure for subsequent threads to find, as well as the processing of blocking queues to ensure the parent/child relationships and thread priority (calculated using the parent thread's priority and a relative priority factor) were current. Update code was modified to ensure that upon completion of the reduction of the closure that the child threads were downgraded to irrelevant, and that in addition to being rescheduled, that the parent threads knew of the evaluation.

Remote operations also required modification. The operations that handled the creation of `RBH` closures and the conversion to `FETCHME` closures required rewriting. Also, operations to provide the handling of messages adjusting priorities, informing of the global addresses of `FETCHME` closures, advising of new parent and child threads, and of thread termination were written.

5 Analysis

5.1 Necessity

The ability for multiple threads to enter the same graph node without contention and conflict is a necessary

component of graph reduction of functional languages since graph components may be shared. The ‘black hole’ locking mechanism utilised in the GUM runtime system [AQUA 1996] enables the first thread to commence evaluation of a portion of the graph while successive threads block awaiting re-awakening by the first thread when reduction is complete. This is an extremely elegant solution. The utilisation of the evaluate-and-die thread creation methodology in which a thread that starts its execution evaluating a particular graph node continues its execution by walking over descendant graph nodes as required, motivates the dynamic thread hierarchy management implementation presented in this chapter. It is the combination of the use of black holes, blocking queues, and the evaluate-and-die model, that gives the thread hierarchy management its success.

The first thread wishing to commence evaluation of a closure does so. Subsequent threads that require the evaluation of the same closure are destined to follow the same reduction path until that closure is reduced. Hence, they are in a very real sense parents of the thread evaluating that closure, just as that thread is a child of the later threads. If all of these threads were able to evaluate the closure simultaneously, the thread with the highest priority would be scheduled to perform the evaluation. This simultaneous reduction is not possible, however, as all but the first thread have blocked upon entering the closure.

In its original state, the GUM runtime system will evaluate the closure as if it is as important as the first thread to reach the closure. Since thread priorities are absent in the original GUM runtime system, whether this thread is more or less useful in terms of overall program execution is a moot point since nothing can be done about it. In the context of the enhanced GUM runtime system which contains priorities, such indiscriminate thread selection coupled with a disregard for subsequent closure entry could result in the closure being evaluated at a much lower priority than it should be. Potentially, a mandatory thread will be blocked on the closure while the first thread to enter it is speculative and is never scheduled! The mechanism presented in this paper increases the effectiveness of speculative evaluation by ensuring that the evaluating thread always possesses the highest priority of all the threads requiring evaluation of that closure.

5.2 Overheads

There are many complexities of dynamic thread hierarchy management: existence of additional threads, closure updating, priority modification, distribution across the processing elements, and thread termination. Each complication is resolved in the extensions to the GUM runtime system proposed here. No additional global indirection closures are required although PARENT, CHILD, and THREAD_GA messages have been added to construct the thread hierarchy. Similarly ZOMBIE and TSO_DEATH messages have been added to deconstruct the hierarchy. A sixth message, PRIORITY, was added to ensure the priority information of the evaluating thread remains current.

If no sharing of closures occurs in the graph, none of these messages will be sent during execution and no blocking queues will need re-awakening. The only overhead of dynamic thread hierarchy management when no such hierarchies exist is the globalisation of the TSO and the storage of a thread’s TSO’s global address in each closure which that thread evaluates. Unfortunately, this cannot be avoided — if the TSO of the first thread to enter a closure is not globalised at that time, when the second thread enters the closure and is added to the black hole, the only way of identifying the original thread is by interrogating each TSO. TSOs, however, are not stored together in a data structure. Given this and the fact that the evaluate-and-die thread creation methodology is used, the first thread may not even be evaluating that closure but one of its descendants instead. Therefore, the overhead is unavoidable, but fortunately, in terms of space, quite low — although it may impact upon garbage collection activity. There is also a size overhead attached to implementing dynamic thread hierarchy management: the addition of space for a global address for every closure. Many closure types do not become involved in the management of thread hierarchies, but the inclusion of the global address in all closures as part of the fixed header is clean, and, as stated, in the case of TSO type closures, simplifies the load distribution extension of thread migration/placement.

With regard to storage, a spark originally occupied one word while a TSO occupied approximately twenty-nine (excluding the info table contents) — some of which are unused. With the enhancements described, a spark occupies seven words while a TSO occupies approximately thirty-two — an insignificant increase. The benefits that occur due to the increase in the size of the spark arguably offset the needed increase in storage size.

6 Conclusions

All modifications to the GUM runtime system are motivated by the goal of improving the speculative evaluation of closures.

All closures have been expanded with a field for storing a global address. This is used to store the global address of the (globalised) TSO for the thread evaluating the closure; this mechanism also serves as a stepping stone for the implementation of full thread placement/migration. A closure may now be interrogated to discover which threads are waiting for its result and this fact allows the evaluating thread to execute at the highest priority of those threads demanding the closure’s evaluation. This information is also available for debugging purposes if required.

Messages have been added to the system to facilitate the management of dynamic thread hierarchies on a multicomputer. The six messages achieve a distributed implementation of such a hierarchy with changes in thread priority elegantly and efficiently performed. Further, the number of messages required to be sent is minimised and acknowledgement messages are not required.

Significant work was also completed to track the relationship between sparks and threads. This required extending the TSO closure type to retain a global address that represented the identity of a parent thread. A similar extension was completed to hold child thread information. This model successfully allowed sparks to move from one processing element to another and not lose track of their parent. In order for this to happen, participating TSOs underwent globalisation (association with a newly generated global address).

One benefit of the presence of parent and child lists within a TSO is the ability to implement priority adjustment. When the value of a graph node is required by multiple threads, the closure evaluation is undertaken at the highest of the priorities of those threads requiring the evaluation. This is possible because of the extensions made here.

All of these additions culminated in the effective management of priorities for shared and unshared closure evaluation and — apart from the expansion of the TSO and closure structure — have little impact upon evaluation if speculative evaluation is not undertaken.

7 Further Work

7.1 Experimental Results

This work developed as one tangential aspect to other work (see [Dermoudy 2002]). Proof of concept has been illustrated and this has world-wide originality. Although implemented, however, no experimentation to specifically test the benefits (and overheads) of the dynamic thread hierarchy management system have yet been undertaken. Such experimentation would be beneficial.

7.2 Thread Migration

The original GUM system supports the allocation of threads yet to be evaluated to remote processing elements. It does not support the migration of a thread that has begun evaluation. The work presented here supports thread migration and the extension of GUM to offer thread migration with dynamic thread hierarchy management is a future challenge.

8 References

Aditja, S., Arvind, Augustsson, L., Maessen, J.-W., & Nikhil, R. S. (1995). Semantics of PH: A Parallel Dialect of Haskell. In *Proceedings of Haskell Workshop, Conference on Functional Programming Languages and Computer Architecture 1995*. ACM.

AQUA Team, The (1996). *The Glasgow Haskell Compiler Version 0.29*. Haskell compiler. FTP site: <ftp.dcs.gla.ac.uk/pub/haskell/glasgow/0.29/ghc-0.29-par-sparc-sun-solaris2.tar.gz>.

Barendregt, H.P. (1984). *The Lambda-Calculus: Its Syntax and Semantics*, North Holland.

Dermoudy, J. R. (1999). Implementing Speculative Evaluation. In *Proceedings of the 6th Australasian*

Conference on Parallel and Real-Time Systems. (pp.355–366). Springer-Verlag.

Dermoudy, J. R. (2002). *Effective Runtime Management of Parallelism in a Functional Programming Context*. PhD Thesis, University of Tasmania, School of Computing.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., & Sunderam, V. (1994). *PVM 3 User's Guide and Reference Manual*. System Manual. Oak Ridge National Laboratory, Tennessee.

Hall, C., Hammond, K., Partain, W., Peyton Jones, S. L., & Wadler, P. (1992). The Glasgow Haskell Compiler: A Retrospective. In J. Launchbury & P. Sansom (Eds.), *Functional Programming, Glasgow 1992* (pp. 62–71). Springer-Verlag.

Hammond, K., Loidl, H.-W., Mattson Jr., J., Partridge, A., Peyton Jones, S., & Trinder, P. (unpublished). *GRAPHing the Future*. Unpublished manuscript.

King, D. J., Hall, J., & Trinder, P. (1998). A Strategic Profiler for Glasgow Parallel Haskell. In the *Proceedings of the 10th International Workshop on the Implementation of Functional Languages*.

Klusik, U., Peña, R., & Segura, C. (2000). Bypassing of Channels in Eden. In G. Michaelson, P. Trinder, and H.-W. Loidl (Eds.), *Trends in Functional Programming* (pp. 2–10). Intellect Books.

Loidl, H.-W. (1998). *Granularity in Large-Scale Parallel Functional Programming*. PhD Thesis. University of Glasgow.

Loidl, H.-W., & Hammond, K. (1994). GRAPH for PVM: Graph Reduction for Distributed Hardware. In the *Proceedings of the 1994 International Workshop on the Implementation of Functional Languages*.

Peyton Jones, S. L. (Ed.). (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall International, London.

Peyton Jones, S. L. (1989). Parallel Implementations of Functional Programming Languages. *The Computer Journal*, 32(2), 175–186.

Peyton Jones, S. L., Clack, C., Salkild, J., & Hardie, M. (1987). GRIP — A High Performance Architecture for Parallel Graph Reduction. In G. Kahn (Ed.), *Functional Programming Languages and Computer Architecture* (pp. 98–112). Springer-Verlag.

Trinder, P. W., Barry Jr., E., Davis, M. K., Hammond, K., Junaidu, S. B., Klusik, U., Loidl, H.-W., & Peyton Jones, S. L. (1998). Low Level Architecture-Independence of Glasgow Parallel Haskell (GPH). In the *Proceedings of the 1998 Workshop on Parallel Functional Programming*.

Trinder, P. W., Hammond, K., Mattson Jr., J. S., Partridge, A. S., & Peyton Jones, S. L. (1996). GUM: A Portable Parallel Implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*.