On Reducing Interprocess Communication Overhead in Concurrent Programs

Erik Stenman
Computing Science Dept.
Uppsala University, Sweden
happi@csd.uu.se

Konstantinos Sagonas
Computing Science Dept.
Uppsala University, Sweden
kostis@csd.uu.se

ABSTRACT

We present several different ideas for increasing the performance of highly concurrent programs in general and Erlang programs in particular. These ideas range from simple implementation tricks that reduce communication latency to more thorough code rewrites guided by inlining across process boundaries. We also briefly discuss the impact of different heap architectures on interprocess communication in general and on our proposed optimizations in particular.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages, concurrent, distributed, and parallel languages*

General Terms

Languages, Performance

Keywords

Concurrent languages, process scheduling, Erlang

1. INTRODUCTION

Large software systems can conceptually be split into several separate and semi-independent tasks. Concurrency tries to provide a convenient form of abstraction for such situations. Hence it is not surprising that many modern programming languages (such as CML, Caml, Erlang, Oz, Java, and C#) come with some form of built-in support for concurrent processes (or threads). Unfortunately many of these languages only provide very crude low-level support for concurrency; for example interprocess communication is often implemented with shared data structures. Promoters of these designs often motivate the low-levelness with the need for speed, using pretty much the same arguments that adversaries of garbage collection for a long time have been arguing for the need for programmer-controlled memory management.

We believe that a higher level of support is needed, not only for memory management, but also for concurrency. A language should provide high-level concurrency primitives and it should be up to the compiler and runtime system to implement these constructs as efficiently as possible.

With a more natural way to handle interprocess communication, such as through explicit message passing, the programmer can concentrate on what to communicate instead of how. This higher level of abstraction does however come with a price: data sent from one process to another is not automatically *there* in some shared data structure to be used by the other process directly. Instead, the other process must also *receive* the data; this typically requires that a *scheduler* prompts the receiving process to access the message and to possibly take some appropriate action. Scheduling and switching between execution environments of processes do come at a cost and in this paper we will present some ideas for reducing this cost.

Our goal is to eventually have truly lightweight processes where message passing is at least as efficient as method invocation in a modern object oriented language.

We have been experimenting with different extensions to Erlang and different designs and implementations within the HiPE system [3], a native code compiler extension to the Erlang/OTP system provided by Ericsson [6]. The rest of the paper begins with a description of some aspects of this system (Section 2). We then present our three main ideas: a *rescheduling send* (Section 3), a *direct dispatch send* (Section 4), and *interprocess inlining* (Section 5). We end with some concluding remarks.

2. ASPECTS OF CURRENT ERLANG IMPLEMENTATIONS

Although some of the ideas we explore in this paper have also been dealt with by the operating systems (OS) community, it is important to note that our context is different since the concurrency in Erlang is not provided by the underlying operating system. Instead the Erlang runtime system itself is assumed to provide much of the functionality often associated with an OS. For example, the runtime system of Erlang/OTP contains its own scheduler, memory manager, code loader, interface to the file system, and an emulator for BEAM code.

Clearly, the implementation of the runtime system will have an impact on the performance of concurrent Erlang programs. Let us therefore describe some aspects of the current implementation.

2.1 Erlang processes

Processes in Erlang are extremely light-weight (lighter than OS threads), their number in typical applications is quite large, and their memory requirements vary dynamically. Erlang's concurrency primitives—`spawn`, "`!`" (`send`), and `receive`—allow a process to spawn new processes and communicate with other processes through asynchronous message passing. Any data value can be sent as a message and processes may be located on any machine. Each process has a *mailbox*, essentially a message queue, where each message sent to the process will arrive. Message selection from the mailbox occurs through pattern matching. There is no shared memory between processes and distribution is almost invisible in Erlang. To support robust systems, a process can register to receive a message if another one terminates. Erlang provides mechanisms for allowing a process to timeout while waiting for messages and a catch/throw-style exception mechanism for error handling.

Note that Erlang processes differ from both OS processes and OS threads: An OS-process usually has a separate address space implemented in hardware resulting in the need of TLB flushes and the like, while OS threads usually communicate through shared memory. Finally, OS processes and threads are often implemented in such a way that they can be executed in parallel.

Erlang processes on the other hand are handled by the runtime system scheduler, which selects a process from a *ready queue*. The selected process is assigned a number of *reductions* to execute, called its *time-slice*. Each time the process does a function call, a reduction is consumed. The process is suspended when the time-slice is used up (i.e., the number of remaining reductions reaches zero), or when the process reaches a `receive` and there are no matching messages in its mailbox.

The scheduler is implemented in C as a function that can be called either by the BEAM emulator or directly from native compiled code. The scheduler takes as arguments the process that has been running and the number of executed reduction steps, and returns the next process to execute.

2.2 Heap architectures

Till the fall of 2001, the Ericsson Erlang implementation had exclusively a memory architecture where each process allocates and manages its own memory area. In this architecture, since each process has its own heap, message passing is implemented by copying the message from the heap of the sending process to the heap of the receiving process. After the message is written, a pointer to the message is inserted into the message queue of the receiving process.

We have implemented a *shared heap* memory architecture for Erlang processes [4, 7], which is already included in the current Erlang/OTP release. Concurrently with that work of ours, Feeley [1] argued the case for a unified memory architecture for Erlang, an architecture where all processes get to share the same stack and heap. This is the architecture used in the Etos system [2] that implements concurrency through a *call/cc* (*call-with-current-continuation*) mechanism. The biggest advantage of a shared heap architecture is that sending a message does not require copying. On the other hand, garbage collection stop times might become longer since all processes share the same heap. A shared heap architecture also opens up some other opportunities for optimizing an Erlang system. We exploit some of them in this paper.

2.3 Behaviours

The Erlang/OTP system comes with the powerful concept of *behaviours*. A behaviour can be seen as an implementation of a design pattern. The OTP library supplies a number of predefined behaviours such as *application*, *gen_server*, and *supervisor*.

With these behaviours the programming of concurrent applications can be taken to a higher level since behaviours supply general solutions to common programming tasks. For example, the programmer does not need to get involved in the details of programming a fault-tolerant server that supports code upgrades.

The drawback of using behaviours is a slight loss in efficiency; since the solutions are general, behaviours tend to employ a number of runtime tests to find the specific solution. Unfortunately there is no formal specification of behaviours, and they are implemented entirely in Erlang. This means that with the current implementation the compiler has no real guarantees about the behaviour of a program that uses behaviours. (The only check that is done is by the linter which gives a warning if any callback function needed by a behaviour is missing. There is no guarantee that the callback function does what it should do, and hence currently the compiler can not trust the behaviour declaration for optimization purposes.)

If behaviours became more formally specified, an optimizing compiler could use the behaviour declaration as a hint on where to look for certain types of opportunities for optimization. For example, an application based on the *gen_server* behaviour does indirect (via function calls) message passing and pattern matching on the message. In essence: all messages to the generic server pass through a call function that tags the message with an atom defining the message type and the server then finds the appropriate handler by pattern matching on that tag. In the abstraction of the behaviour the information of the message type is lost, and it can not be found by e.g. conventional partial evaluation since message passing is involved.

This common pattern was actually one of the inspirations to interprocess inlining; we feel that users should be able to use this powerful behaviour without worrying about loss in performance.

3. RESCHEDULING SEND

Interprocess communication in Erlang is asynchronous, and the `send` operation is non-blocking. However, these are actually conceptual aspects on the language level, and there are several ways to implement them in the underlying runtime system.

The current Erlang system is implemented in the natural way, that is, the `send` operation just places the message in the receiving process' mailbox and then the sending process continues executing until it either blocks in a `receive` statement or has exhausted its time-slice.

In most cases, when a process sends a message it is because the application wants the receiver of that message to act upon the sent information. Hence, it would probably be in the best interest of the sender to yield to the receiver in this case, and let the receiver act on the message. We will refer to this type of `send` as a *rescheduling send* operation.

We therefore propose to implement this by letting the `send` operation, at least in some cases, also suspend the sending process. This would hopefully lead to lower message passing latency since the receiver can start executing directly when a message is sent. We also expect that in many cases the cache behavior would be better since the receiver will get the message while it still is hot in the cache.

In a private heap system, since the message has to be copied, the whole message should be hot in the cache right after the `send`. Hence, it is important to directly switch to the receiving process before the sender starts producing new data. In a shared heap system, the message does not need to be copied but as it is likely to have been created recently, it is also likely to be hot in the cache.

The real benefits of this design will probably depend both on the underlying hardware and on the communication characteristics of the Erlang program. We do not believe that the benefits of this optimization will be very significant in isolation, but the ability to suspend a process directly after a `send` can open up possibilities for further optimizations.

4. DIRECT DISPATCH

The idea to let the `send` operation suspend the process can be taken one step further by completely bypassing the scheduler. Since it is often the case that the sender is suspended waiting for the receiver to react on the sent message, a natural action for the sender to take is to contribute its remaining time-slice to the receiving process hoping that this will lead to a faster response. We therefore propose a *direct dispatch send* operation: After `send` has placed the message in the mailbox of the receiver, any reductions left could be passed to the receiving process, which could be woken up directly (ignoring the ready-queue).

With this approach, some overhead of the scheduler could be eliminated and the latency of message passing would be reduced even further. Since this approach would also guarantee that it really is the receiver of the message that will execute next, the effects of having the message in the cache will hopefully also become more evident.

As with any process, the receiver is allowed to execute until it blocks in a `receive`, or the reduction count reaches zero, or it performs a direct dispatch send of its own. If the receiver was taken from the ready queue and becomes suspended because any of the two latter reasons (i.e. it is still runnable), it is important to reinsert it into the ready queue in the same position as it was taken from, lest it might starve.

If the receiver performs a direct dispatch send back to the original sender then that sender can get back the remaining reductions and can keep on executing as usual. This way the common case, where one process sends a request to another and then receives a reply to the request, can be almost as efficient as a function call.

5. INTERPROCESS INLINING

To take these optimization ideas even further, we would like to not only change the behavior of the send operation in the runtime system, but actually optimize the code executed before a send and after the accompanying receive. The goals of this optimization are to reduce the overhead of message creation (for example, by avoiding enclosing parts of a message in a tuple), reduce context switching overhead, and open up possibilities for further optimizations by considering the code of the receiver in combination with that of the sender.

The optimization is performed on a pair of functions, the function containing the `send` and the function containing the `receive`. We will refer to these functions as f and g respectively, and the pair as a *candidate pair*. The code at the point of the `receive` statement in g is inserted into the code of f at the point of the `send`. The resulting code is then optimized using standard compiler optimization techniques.

To perform this optimization we have to respect the following requirements:

1. Find a program point where a `send` is performed.

2. Find out at which `receive` statement this message is received.

3. Ensure that, at the time of the `send`, the receiving process is suspended at the `receive` statement found in step 2.

4. Ensure that the mailbox of the receiving process is empty.

Since this process communication behavior can be hard to analyze statically — in any concurrent language and in a dynamically typed language such as Erlang in particular — we propose the use of profiling and dynamic optimization to implement this interprocess code merging.

To do this we take advantage of two features of Erlang: *hot code loading* and *concurrency*. The presence of concurrency makes it possible to implement supervision and recompilation in processes in a way which is separated from the application. Support for hot code loading ensures that there are methods for linking and loading re-optimized code into a running system in an orderly way. We also use a special HiPE extension that makes it possible to replace code on a per function rather than on a per module basis.

We first instrument the system in order to profile the aspects that can trigger a recompilation. During normal execution a supervisor process monitors the profile. When the profile indicates that a part of the program should be recompiled, the supervisor starts a separate process for the compilation.

The gathered profile information is then used to choose candidates for inter-process optimization. These candidates consist of pairs of program points; one program point refers to a `send` statement, and the other refers to the corresponding `receive` statement. These pairs are found by profiling each send to collect information during execution. The collected information has two components: information about the destination (*Dest*), and the number of times the instruction is executed (*Times*). The *Dest* field is initialized to *none*, and the *Times* field to zero. When the send is executed, the *Times* field is increased and the receiving process is checked. If the mailbox of the receiver is empty then the program counter (PC) of the receiver is checked; if the PC is equal to *Dest* or if *Dest* is equal to *none* then *Dest* is set to PC. Otherwise *Dest* is set to *unknown*.

In the case where a `send` has only one `receive` destination, the `send`/`receive` pair is considered as a candidate for the

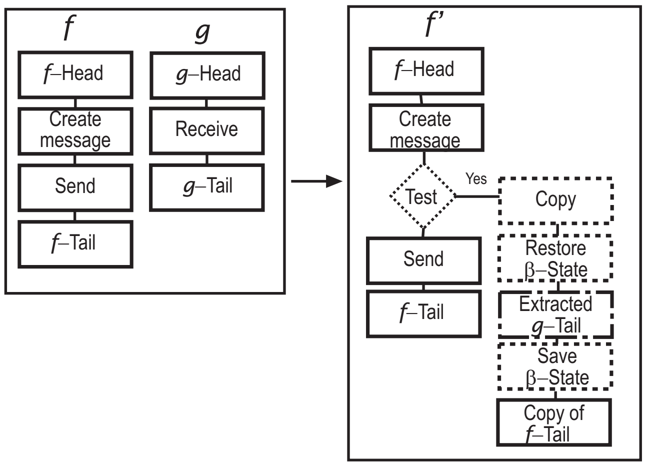

Figure 1: Before the merging, function f is executed by α and function g is executed by β. After the merging, f' is executed by α.

optimization. On the other hand, if a `send` has more than one destination, even if it is just two different destinations, then the profiler will classify the `send` as *unknown* and no optimization will be performed.

When a frequently executed candidate pair is found, the functions containing the `send` and the `receive` are compiled to intermediate code. The intermediate code fragments of the two functions are then merged. In short, the merging is done so that the program point of the `send` statement is connected with the program point of the `receive`. The resulting code is optimized and compiled to native code.

To ensure correct behavior, execution of the optimized version is guarded by a run-time test. This test checks that requirements 3 and 4 in the above list hold; otherwise the original unoptimized version is executed.

5.1 The transformation

We will refer to the sender (the process executing f) as α and the receiver (the process executing g) as β.

For a given `send`, the function f can be divided into the following abstract blocks of code:

1. Head (code preceding the `send`)
2. Message creation
3. `send`
4. Tail (the rest of the code)

The function g is divided into:

1. Head (code preceding the `receive`)
2. `receive`
3. Tail (the rest of the code)

The intention of the transformation is to allow process α to execute code that would otherwise have been executed by process β. Thus, the resulting code for α, function f', will contain fragments of the code from g; see Figure 1.

The merged function f' is a copy of the function f with these six additions:

1. Test — A test is inserted before the `send` in f'. This test checks whether β is suspended at the right program point (at the `receive` in g) with an empty mailbox. If this test succeeds the execution continues with the optimized code (item 2), otherwise the execution continues with the original code of f.
2. (Message copying) — In a system with a private heap architecture the message is copied from the heap of process α to the heap of process β using an explicit copy operation. (In a shared heap system, no copying is needed.)
3. Restore state — All live β–temporaries are read from the stack of β. (This is done by consulting a mapping from intermediate code temporaries to stack positions.)
4. Code from g — The code from g that is suitable for external execution is then executed.
5. Save state — All live β–temporaries are written back to the stack of β.
6. f–tail — A copy of the tail of f is executed.

Since we rely on a subsequent optimization pass to clean things up, performing the merging is straightforward. The subsequent optimization pass, which performs a *generalized constant propagation* and *dead-code elimination* [5], will remove unused paths from g.

In our context, the generalized constant propagation propagates not only true constants, but also Erlang terms such as lists and tuples with dynamic elements. The propagated information is then used to fold tests and element extractions on these structures. When the tests are folded and short-circuited, we perform dead code elimination and removal of unreachable code.

Often in Erlang, parts of the messages are just used for switching on the type of message. Interprocess optimization together with generalized constant propagation helps us avoid the copying of these parts of the message.

The code from g has to be rewritten so that it can be executed “externally”, that is, from within process α. This means that the primitives we want to inline have to be rewritten for external execution.

We can extract almost all instructions from g for merging with f, as long as the code fulfills four prerequisites:

1. There has to be some way of ensuring that we do not get code explosion.
2. The code may not suspend.
3. The control flow may not be passed to code that is not adapted to external execution.
4. The extracted code must terminate otherwise process α might hang.

To make sure that these prerequisites are fulfilled some instructions are not extracted:

1. A `call` to another function, a meta call (`apply`), or a `return` can not be extracted since the control could be passed to code that is not adapted for external execution.

2. Instructions that lead to the suspension of the process, such as the explicit suspension instruction or a `receive`.

3. Some built-in functions are large and uncommon and not worth the effort to adapt for external execution.

4. Non-terminating code is unacceptable. If some bug in β makes it loop forever, we do not want this bug to propagate to the process α. To ensure that the extracted code terminates, we do not accept any loops in the control flow graph of the extracted code. Note that this is not such a harsh restriction as it may sound, since the only way to get a loop in the intermediate code is by making a tail-recursive call where the caller and the callee are the same. If there is a loop it will probably contain the `receive` that caused the extraction in the first place. In this case the control-flow graph will be cut at this point and the loop will be broken.

The instructions in the g–tail that do not belong to any of the categories listed above are extracted. A control flow path that contains an instruction that is not extractable is cut just before that instruction.

To propagate changes in the state of β we have to save the new state at the end of the extracted code. To this end, we write all live temporaries back to the stack at the end of each path of the extracted code. At the end of each of these paths, the continuation pointer of β is set to point to a stub containing the instructions from that path that could not be extracted from g.

To simplify optimization we duplicate the tail of f. At the end of each path of the extracted control flow graph we insert a `jump` to this copy. This ensures that when the code in the copy is reached, the execution is guaranteed to have passed through the code extracted from g.

5.2 Further considerations

In a runtime system architecture where each process allocates its private heap, the garbage collector typically relies on the fact that all data structures accessed by a process are allocated on the heap of that process. This invariant is temporarily broken while the process α accesses the state of process β, but since we have control over when α is suspended and when garbage collection is triggered, we can ensure that the invariant is maintained at these points. In a shared heap architecture, this is not a problem.

Our inter-process optimizer will change the scheduling behavior. One might suspect that this could lead to a change in the concurrency semantics of the program. However, note that since in the optimized code we do not allow the code from g to loop and count each reduction that would have been counted before the optimization, the observable behavior will remain unchanged.

The inter-process optimizer will merge code from two functions (f and g). If the module of g is updated with hot-code loading, old code from g will remain inside f (actually in f'). However, this code will never be executed, since the runtime test in f' only succeeds when the receiver is suspended from old code. (If the module containing f is replaced then all optimized code is removed and in this case there is no problem at all.)

5.3 Return messages

The situation where the receiver of a message sends a message back to the original sender is so common that we have decided to handle this situation specially. The technique we have devised requires the following criteria to be fulfilled:

1. There is a `send` in g–tail.

2. The destination of the `send` in g is the process α.

3. All paths through f–tail contain a `receive`.

4. The mailbox of α is empty.

We ensure these criteria by first of all always check that the mailbox of α is empty before we use the optimized code. By doing this check in the beginning, we get a very simplified control flow graph for f'.

In a private heap system we just copy the message from the heap of process β to the heap of process α, if the destination of the `send` is α. In a shared heap system no copying is needed, the pointer to the message can be put directly in the temporary containing the received message. Now, the nice thing is that by using *generalized constant propagation* we can often remove the runtime tests completely. Depending on how the message is used, we might also get rid of the copying between the processes completely even in a private heap architecture.

5.4 Potential gains

With interprocess inlining we can reduce the overhead of process communication in four different ways:

1. **Short-circuit switches on messages**
   We can use the information about the form of the message to short-circuit the pattern matching in the `receive`. Since the switching usually is made up of several tests on heap allocated data, short-circuiting results in a control flow path with fewer `load`, `compare`, and `branch` instructions.

   We also expect that this will also make the hardware prefetching mechanisms work better. If the receiver can receive several different messages that have the same frequency, then the switch will go in different ways each time rendering the prediction useless, which results in pipeline stalls.

2. **Reduce message passing**
   It is quite common in Erlang programs that a process creates a message, sends it to another process, which subsequently performs some matching on the structure of the message, accesses some components of the message and never looks at the whole message again.

   By short-circuiting switching on the message we can avoid the creation of the message (and also save time in the garbage collector).

3. **Reduce context switching**
   We can, in the cases where the receiver immediately answers, remove the context switch completely. This not only means that the receiver does not need to be scheduled, but it also means that the executing process does not need to be suspended. Measurements indicate that in many concurrent Erlang programs the processes do not exhaust their time-slice but they are

instead suspended on `receive`. If the sender can keep on running until the time-slice is used up then the expensive scheduler would be executed less. Letting the same process execute longer also results in better cache behavior.

4. **Enabling of further optimizations**
The most significant gain can come from the ability to do optimizations on the merged code, just as the real gain from procedure inlining comes from the optimizations done after the inlining. We get the possibility to do, for example, constant propagation, common subexpression elimination, and register allocation, on merged code from the sender and the receiver.

6. CONCLUDING REMARKS

We have presented several different methods for cross-process optimization aiming to reduce the overhead for interprocess communication. These methods also enable further optimizations across process boundaries, such as constant propagation and more global register allocation. The context switch can be completely eliminated in some cases, reducing the overhead for concurrency.

These optimizations will speed up existing Erlang programs without requiring any modifications to the source code. Since the use of processes will be less expensive, the usefulness of concurrency is extended, making it possible to use processes in cases where it previously has been considered too expensive.

Acknowledgments

We thank Sven-Olof Nyström for interesting discussions on the implementation of interprocess inlining. This research has been supported in part by the ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson Development.

7. REFERENCES

[1] M. Feeley. A case for the unified heap approach to Erlang memory management. In *Proceedings of the PLI'01 Erlang Workshop*, Sept. 2001.

[2] M. Feeley and M. Larose. Compiling Erlang to Scheme. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, number 1490 in LNCS, pages 300–317. Springer-Verlag, Sept. 1998.

[3] E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, Sept. 2000.

[4] E. Johansson, K. Sagonas, and J. Wilhelmsson. Heap architectures for concurrent languages using message passing. In *Proceedings of ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, pages 88–99. ACM Press, June 2002.

[5] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufman Publishers, San Fransisco, CA, 1997.

[6] S. Torstendahl. Open Telecom Platform. *Ericsson Review*, 75(1):14–17, 1997. See also: http://www.erlang.se.

[7] J. Wilhelmsson. Exploring alternative memory architectures for Erlang: Implementation and performance evaluation. Uppsala master thesis in computer science 212, Uppsala University, Apr. 2002. Available at http://www.csd.uu.se/projects/hipe.