Uppsala Master's Thesis In
Computing Science 89
Examensarbete DV3
1998-06-02
ISSN 1100-1836

A Parallel and Multithreaded ERLANG Implementation

Pekka Hedqvist
June 2, 1998

Computing Science Department, Uppsala University
Box 311, 751 05 Uppsala, Sweden

This work has been carried out at:
Computer Science Lab
Ericsson Telecom
ÄT2/ETX/DN/SU
SE-126 25 Stockholm, Sweden

Supervisor: Tony Rogvall
Examiner: Johan Bevemyr

Passed:

1. Introduction 3

1.1. Noteworthy topics 3

1.2. Acknowledgements 4

2. The ERLANG System 5

2.1. ERLANG 5

2.2. Major ERLANG features 5

2.3. Modules, Functions and Clauses 6

2.4. Data Objects 6

2.5. Variables and Pattern Matching 6

2.6. Concurrency 6

2.6.1. 6

2.6.2. 7

2.7. Error handling 7

2.8. Dynamic code handling 8

2.9. Scheduling 8

2.10. Ports 8

3. ERLANG Runtime Systems 9

4. Parallel Computers and Programming 11

4.1. Parallel computers 11

4.2. Parallel and concurrent programming 11

4.3. Threads 12

4.3.1. Implementation 12

4.3.2. Posix standard 12

4.3.3. Win32 12

4.3.4. Portable threads 12

4.4. Memory ordering on multiprocessors 13

5. The Multithreaded ERLANG Implementation 15

5.1. Introduction 15

5.2. Implementation description 16

5.2.1. Process interaction 16

5.2.2. Memory management 17

5.2.3. Tasks 18

5.2.4. Scheduler 20

5.2.5. Threads 21

5.2.6. Task Priority 22

5.2.7. I/O and timer 22

5.2.8. Tasks and threads 23

5.2.9. Dyncamic code handling 24

6. Measurements 27

7. Conclusion 31

References 32

Abstract

This masters thesis describes the design and implementation of, and the experiences with a multi-threaded and parallel runtime system for ERLANG.

ERLANG is a declarative language with real-time properties, processes, fault-tolerance, distribution and dynamic code handling. It is very suitable for large complex control systems with fault-tolerance demands. The goals of this implementation are to obtain scalability with multiprocessor hardware, improve real-time properties, keep a portable and unified code base with the non-threaded system. Covered areas include major changes in the system from the previous, design decisions for system throughput, parallelism, real-time properties, general improvements and benchmarks.

ERLANG was developed at the ERICSSON Computer Science Laboratory and is now gaining wide acceptance within ERICSSON. It is in widespread use since several years and is now a mature technology. ERLANG is the primary development language for several commercial core products and is used in a number of current product development projects at Ericsson. It has also gained interest within the academic world and is now marketed externally.

The thesis starts with a motivation of what a multi-threaded runtime system can do for ERLANG and typical applications. Then a description of the implementation follows and various aspects of the systems are benchmarked and discussed.

Keywords: multi-threading, parallelism, symbolic programming, concurrency, real-time systems, runtime systems.

1 Introduction

ERLANG [1] evolved during several years of research in the eighties and nineties in search for a better way of building complex control systems with requirements on distribution, robustness, availability and soft real-time, as is common in the telecom and datacommunications industry. It is now an ERICSSON product[1] and is used in a number of industrial product development projects [3]. An Erlang system is typically accompanied by a number of software tools, system architecture support libraries and databases [2] .

ERLANG has lightweight processes, asynchronous message passing and no explicit shared memory. This allows parallel and concurrent applications to be created without many of the hazards connected with shared memory programming and its synchronization primitives. But the current ERLANG runtime system can not let ERLANG processes execute in parallel on parallel hardware - a highly desired property if one is to create scalable, load tolerant and high performance systems. One way to overcome this is to run several ERLANG nodes as separate system processes on a parallel computer and use the ERLANG distribution facility. This however introduces problems like high communication overhead between ERLANG processes on different nodes and load balancing difficulties - the application must decide on which node to start new ERLANG processes and it is difficult to migrate running processes between nodes.

By executing ERLANG processes with the help of an operating system thread library, advantages like parallelism and improved real-time properties can be achieved. Load balancing between processors is managed by the operating system among several other things. This however needs a thoroughly modified and redesigned runtime system; it also introduces potential problems of more heavyweight ERLANG processes and a system which behaviour varies more depending on the operating system.

1.1 Noteworthy topics

ERLANG applications usually has many ERLANG processes. Since only asynchronous message passing exists and the application domain is often naturally parallel - many processes execute code, receive and pass messages and go on executing code. By this behaviour one can assume that several existing applications would benefit from a parallel platform. By making parallelism an issue during application development one could expect even better results from parallelism on a broad range of ERLANG application domains with little extra effort. But it is important to keep in mind that this is highly dependent of the implementation itself and the application domain. One way to determine the potential parallelism in existing applications without real-life testing is to poll the internal ERLANG run queue on regular intervals. If the average length is above zero there is processes that can execute in parallel and it indicates the level of processors that can be efficiently used. This truncated number minus one gives the amount of processors that can be utilized in parallel. From the peak queue length the maximum potential parallelism in the application can be calculated.

The lack of explicit shared memory handling in ERLANG makes it difficult to create some really efficient parallel programs on shared memory parallel computers with a parallel implementation. This limitation is known from other areas, ERLANG is not optimal for all applications.

1. Contact ERLANG Systems of Ericsson Software Technology for product information

The current ERLANG implementation provides very fast context-switch times. A thread-based implementation would probably not provide better switch times and would make the system more dependent on the hosts scheduling algorithm. A good thing would be that any features on the host system would ERLANG take advantage of, like for example a defined scheduling algorithm with hard real-time properties.

The garbage collector used in ERLANG [1] cannot be interrupted when started, the whole heap must be collected. This is usually not a big problem; since each process has a separate heap that is garbage collecteded individually which results in small collections with little program execution interruption. But a collection of a big heap takes time, this can have severe impact on the real-time characteristics of some applications. If ERLANG processes ran on system threads context switches can occur at any time [2].

Another desirable property would be the possibility to create a well-defined interface for executing foreign code as threads within the runtime system, and be able to treat them as ERLANG processes.

Foreign code that is linked with the ERLANG runtime, or calls to blocking system libraries, can easily break the whole runtime system. This can not be totally avoided in a threaded runtime system since they share the same adress space, but a range of faults can be eliminated since the internal scheduling mechanism often is truly preemptive. With a defined internal process API it would also become feasible to create "ERLANG" processes in any language, processes that looked and behaved like normal ERLANG processes.

1.2 Acknowledgements

Tony Rogvall of the Ericsson Computer Science laboratory was the supervisor of this work; he is the main responsible for the creation of this prototype. I am grateful for all his help and guidance. Many thanks to *Tony* and also to all the CS lab members. Also thanks to *Johan Bevemyr*, my examiner, for his support during this period.

1. Except in the VEE implementation that has an incremental GC. VEE is not generally available
2. If the thread library allows context switch during compute bound execution, otherwise some thread *yield()* calls can be done at regular intervals

2 The ERLANG System

2.1 ERLANG

The Computer Science Laboratory at Ericsson developed ERLANG. The ERLANG Systems division of Ericsson Software Technology AB markets ERLANG and consults in projects both within and outside the Ericsson sphere. Implementations of ERLANG can be obtained free of charge subject to a non-commercial license.

ERLANG evolved after several years of experiments with different existing languages in search for one suitable for large industrial tele- and datacommunication applications. No “perfect” language for this existed and experiments with a new language begun. These experiments resulted in ERLANG and a whole middleware system, OTP[1], based on it.

ERLANG is basically a untyped functional language like ML and LISP with a Prolog inspired syntax combined with features found in real-time languages like process constructs, message passing and advanced error handling. Powerful support for distributed programming is also included in the language.

A brief introduction of ERLANG follows here. Further reading can be found in the ERLANG book [1].

2.2 Major ERLANG features

- **Language** - Declarative syntax with dynamic typing, functional semantic largely free from side effects. Declarative languages are often regarded as very expressive, resulting in readable and compact code. Properties often associated with few bugs and productive programmers. Several features common in other declarative languages were omitted to make it more suitable for real-time behaviour and control of hardware and to keep the language and implementations simple. Omitted mechanisms include: *backtracking, currying, lazy evaluation* and *deep guards*.
- **Concurrency -** Lightweight processes with asynchronous message passing are integrated into the language. There are mechanisms for allowing processes to timeout while waiting for messages and to read a real-time clock.
- **Real-Time -** Soft real-time demands in the millisecond range can be handled.
- **Robustness** - Several constructs to catch and handle run-time errors and monitor other processes, even through a network.
- **Code Management** - Dynamic code management primitives are integrated to allow code replacement in running systems.
- **Memory management** - Real-time garbage collection automatically manages memory reclamation. No explicit allocation and deallocation is needed, no explicit pointers exists.
- **Distribution** - Primitives to create and manage ERLANG nodes in a network. Transparent message-passing and process control between nodes.

1. Open Telecom Platform

2.3 Modules, Functions and Clauses

ERLANG source code is composed by a number of modules, which consists of functions built by clauses. Functions must be exported to be seen outside the module and are by default local to it. The following example defines a known program:

```
-module(example).
-export([ackerman/1, ack/2]).
ackerman(X) when X >= 0-> ack(X, X).

ack(0, Y) -> Y+1;
ack(X, 0) -> ack(X-1, 1);
ack(X, Y) -> ack(X-1, ack(X, Y-1)).
```

The annotation **`-module(Name)`** denotes the module name that must be the same as the filename without the **`".erl"`** extension and **`-export(ListOfFuns)`** is the functions this module exports, functions not exported can not be called from outside the module.

2.4 Data Objects

ERLANG is a dynamically typed language. Variables begins with a upper-case letter. There are **constants, atoms, integers, floats, pids, refs, binaries and functional objects**. And *compound terms*: **tuples** for a fixed number of objects and **lists** for a variable number of objects.

2.5 Variables and Pattern Matching

A main concept in ERLANG is pattern matching which is used in a number of contexts like function head matching, if, case and receive statements, in variable assignment etc.

When a function is called one of its clauses is chosen like in Prolog. In the example a call **`ack(4,0)`** would match clause two of **`ack/2`** since the first argument would not match clause one and two. Since there is no backtracking the fourth clause would never be called even tough it also matches with the call. Pattern matching is also used to assign variables. Variables can only be assigned once in the same scope.

Example:

- **`Foo = bart`** - Succeeds, Foo is bound to bart.
- **`{A, B} = {4711, bar}`** - Succeeds, A is bound to 4711, B is bound to bar.
- **`{C, C} = {42, 42}`** - Succeeds, C is bound to 42, then matched with 42.
- **`{C, C} = {13, 42}`** - Fails, C can not be bound to two different values.

Note that unlike prolog there are no logical variables. Unbound variables can only exist on the left side of the assignment.

2.6 Concurrency

2.6.1

Concurrency is a central part of ERLANG. It is provided through ERLANG processes which are light weight and cheap to create and terminate. Processes communicate through asynchronous message passing. There is no guarantee of delivery, but messages are guaranteed to arrive in the same order as they were sent. Any synchronization is implemented in the application.

A process has a mailbox where messages are stored. When a process enters a receive statement it will look in turn at each message and to match each pattern with the message. A matched message is removed and the corresponding sequence is evaluated. An optional **after** construct introduces time. If no received message is matched after a specified amount of milliseconds its sequence is evaluated.

A small example:

```
-module(watch_counter).
-export([loop/1]).
loop(Val) ->
   receive
       increment ->
          loop(Val + 1);
       {From, value} ->
          From ! { self(), Val},
          loop(Val);
       stop ->
          true;
       Other ->
          loop(Val)
   after watchdog_timeout_interval() ->
          error_logger ! {watchdog_timeout,self(), Val},
          loop(Val)
end.
```

Last call optimisation is always performed on iterative like code to ensure that the stack does not grow. The "**!**" sign is the send operator. A typical piece of code that does initialization and uses the counter could look as follows:

```
...
Pid = spawn(counter, loop,[0]).
Pid ! increment.
Pid ! increment.
Pid ! {self(), Value}, receive {pid , Value} -> Value end.
...
```

A process can have an associated atom name to allow other processes to communicate with it without having its process identifier.

Distributed processing is transparent. When a process is spawned an extra argument can be added to indicate on which node is should be created. The process identifier can be used in the same way as local ones.

2.6.2

A noteworthy property of ERLANG is that processes are protected from each other even if they are implemented within the same address space in the virtual machine[1]. Terms are always complete since there is only asynchronous message passing, there cannot be ''half'' or uninstantited terms.

2.7 Error handling

Function call error handling in ERLANG is implemented using a dynamic catch/throw mechanism.

1. Assuming of course that the virtual machine is correct

If any call evaluates to some error without a catch the ERLANG process executing the code terminates. Processes can monitor each other's behaviour. This can be described in terms of two concepts, process *links* and **'EXIT'** signals. A process can establish links to other processes. If a process terminates for any abnormal reason a special exit signal is sent to all processes and ports that are linked to the process. On receipt of this signal the default action is to terminate the process and propagate further exits signals to all processes linked to this process. By setting a process local flag **trap-exit** to true the exit signals are received as normal ERLANG messages in the message queue, with the format **{'EXIT' ExitPid, Reason}**. The program can then take any desired action when receiving exit messages. If **Reason** is the atom **kill** the process is unconditionally terminated. The functions **link(Pid)** and **unlink(Pid)** are used to establish and remove bi-directional links between processes.

Processes on different nodes in a network can be linked; an ERLANG node can also be monitored explicitly.

2.8 Dynamic code handling

ERLANG supports code replacement during execution of ERLANG code. Modules can exist in at most two versions, *new* and *old*. If a newer version of a module is loaded all new function calls that have *explicit module* references refer to *new* code. Other calls continue to use the *old* code.

2.9 Scheduling

Scheduling of ERLANG processes is an implementation-dependant issue but some criteria must be fulfilled by all implementations. Process scheduling must satisfy the property of fairness; that is, any process than can be run will eventually be run. No processes is allowed to block other processes for a long period. Note that this differs from what many thread libraries or preemptive priority based operating systems provides, in which a thread executing compute-bound code runs until completion before another can start execution if there is only one processor in the system.

2.10 Ports

Since ERLANG is process oriented *ports* are used to represent the external non-ERLANG environment. Creation, termination and communication with ports are similar to processes. Ports represent file descriptors, system processes, hardware device drivers, linked in C code etc. Ports are created with a call to **open_port(Args)**, which returns a port identifier similar to a process identifier. There is always a process owning the port, usually the creator. But several processes can link, communicate and receive exit signals from and with it.

3 ERLANG Runtime Systems

The first ERLANG prototype implementation was created in Prolog. Today's ERLANG consists of a compact and highly optimised runtime system written in C [4] and a large number of libraries written in ERLANG. The externally available implementations are JAM[1] and BEAM[2] [6]; they share a common runtime system but differ in how they execute code. JAM is a byte-code interpreter. BEAM can compile ERLANG code to C, that is compiled to native code and to threaded code. Process heaps are separate and are independently garbage collected with a modified two-generation semi-space stop and copy algorithm. Both stack and heap can grow and shrink as needed. The cost of message passing grows with message size, garbage collection is usually fast and interrupts the system minimally. This design gives a simple and efficient implementation.

There are also experimental implementations of ERLANG; one is VEE[3] [5] that mainly differs in its heap and message passing handling. It uses a unified heap for all processes and has an incremental generation garbage collector. Even tough the cost of message passing is constant, the overall system performance is not good enough yet..

Except the implementation presented in this report, all ERLANG implementations so far has had internal scheduling without pre-emption. Either the compiler is known to generate code which at regular intervals decreases a counter or the emulator loop does this. When this counter reaches zero the system inserts the currently executing process in a round-robin run-queue and fetches the next process ready to execute. This gives very effective time slicing and minimal overhead for context switching. The cost for updating the counter is minimal since its granularity is fairly coarse. Since no unexpected pre-emption can occur the implementation can be somewhat optimised.

1. Joes Abstract Machine
2. Bogdans ERLANG Abstract Machine
3. Virdings ERLANG Engine

4 Parallel Computers and Programming

4.1 Parallel computers

Multiprocessor machines of today are expanding from specialized high-end markets to more commodity hardware. Nearly all of them provide one logical view of memory although they can be highly distributed internally. Tightly connected systems with a shared memory or distributed systems with distributed memory, or in other words, shared all or shared nothing systems. Hardware and operating system transparently provides one logical view of memory for application programmers. Not discussed in this report are parallel computers with only explicit message passing between CPUs, i.e. specialized distributed high-speed computer networks [7].

The global memory machines often use a mix of several implementation technologies. Processors has several levels of cache, they can be interconnected through a traditional shared memory-bus, a network-based bus[1] [9], a proprietary network with some logical topology or a crossbar switch [10] etc. In some implementations there is no central memory[2] [8], other has central memory[3]; in all cases there is some memory coherency protocol keeping the memory consistent.

Today the bus-based variant is common in the low-end market that is dominated by Pentium Pro and Pentium II based systems. The border between low, middle and high-end systems is fuzzy since the previously low-end machines now can sometimes cross into the higher-end machines performance league.

4.2 Parallel and concurrent programming

There are many ways of programming parallel and concurrent systems. Some programming languages lacks any support for concurrency and others have it as an integrated part of the language. Common paths are threads or processes with shared memory and different synchronization primitives, or processes with message passing. Which is best is often a matter of debate and personal preferences, but it also depends on the programming domain and on what computer architecture the application will run on.

Compilers can parallelize sequential code, but their commercial use is limited to mainly mathematical number crunching applications. For a large number of applications it is not a compute problem with an algorithm that is to be solved in parallel. It is often a number of parallel tasks that cooperate more or less intimately between themselves and the outer world like many database and telecom applications.

It is nontrivial to fully exploit the potential of parallel computers. The view of memory as one logical unit is in many ways convenient, but can also lead to many programming errors. Memory bandwidth is usually high but the worst case latency can be *very high*, so memory reference locality is important, so is avoiding of memory contention between processors. The added latency from each level in the memory hierarchy is often in the order of one magnitude. Sometimes it can be effective to program applications with explicit message passing and copying, in other cases data sharing and locking is better. In any case it is vital to understand the implications of these factors to be able to make good use of todays parallel computers [12].

1. SCI scalable coherent interface, FireWire or some other network bus
2. ccNUMA - cache coherent Non Uniform Memory Architecture
3. Non Uniform Memory Architecture

4.3 Threads

Using multiple threads of execution to express concurrency and parallelism is not new [17]. ERLANG has an internal scheduling mechanism that gives at least the same light-weightiness as a thread library, but the ERLANG language makes it impossible to destroy or disturb other ERLANG processes. Threads are traditionally programmed in C or C++ without any protection between different threads.

Today most operating systems that runs on parallel hardware has threads as an effective method for using this hardware since they share memory and are cheap to create, destroy and to context-switch between. Before threads were common, processes were used; they communicated with pipes or shared memory. This gave unnecessary overhead when memory protection and fair scheduling were not needed, or maybe more common, was too costly in terms of performance.

4.3.1 Implementation

Threads can be implemented in several different ways on an operating system. They can be implemented purely in user level, with simulated concurrency in one process. This method does not make it possible to run parallel threads on different CPUs. Another implementation method is to extend the kernel with an API that provides the process with ways to create schedulable entities in the OS kernel; without a new address space and other costly features. These kernel entities can execute code independently. They have a set of lightweight synchronization primitives, are relatively cheap to context switch between and can run on several CPUs in parallel. To achieve even more light weight threads mixed implementation solutions exists like; two level scheduling [20][13][14] with user level threads multiplexed on a pool of kernel threads, or threads with upcalls from the kernel.

4.3.2 Posix standard

Recently a Posix threads standard was accepted [15]. It is or will be implemented by most Unix and real-time OS vendors. It had the DCE[1] and UI[2] threads as its foundation but is now, when finished, very much extended and refined. The standard gives great freedom on how threads are implemented below the API as long as the specification is followed. It is also quite broad and several parts are optional for a compliant implementation. For instance, there are several scheduling policies defined, including one that can be vendor specific. A vendor must implement at least one or several of these.

4.3.3 Win32

The Win32 API has its own thread API [16] that is separate from the Posix standard, some synchronization services also differs in its semantics from Posix. This departure from standard integrates the Win32 threads more nicely with the rest of the Win32 API but creates extra effort to make software portable.

4.3.4 Portable threads

The lack of a thread standard has resulted in a number of different thread APIs and semantics; usually between platform but sometimes even within a platform. Several thread middleware libraries exists to ease this problem [18][19]. The Posix standard will probably solve much of

1. Distributed Computing Environment
2. Unix International - mainly Suns Unix flavor

this in due time, but Win32 will not comply with the standard, to keep portability at the C and C++ level there is still a need for cross platform middleware; either internally developed, or one of the many free or commercial middlewares.

4.4 Memory ordering on multiprocessors

Before the nineties multiprocessor machines usually had strongly ordered memory [11]; when one processor makes two writes to two different addresses other processors sees these writes in the order they were performed. But with more scalable systems the trend is toward relaxing memory order. So called weakly ordered memory is now common; a write to memory by one processor is not necessarily available immediately to another processor. When one processor makes two writes to two different addresses other processors do not necessarily see these writes in the order the first processor performed them. Even ''obviously'' correct code can break. Classical synchronization must be complemented with primitives to ensure that cache store barriers are are inserted before synchronization points are passed. This is usally included in the syncronization primitives. Low-level optimisations must be very carefully tailored and are platform dependant.

5 The Multithreaded ERLANG Implementation

5.1 Introduction

A multithreaded, or maybe more correctly a parallel ERLANG implementation must fulfil several properties:

- **Reentrant and thread safe** - removal of all unnecessary global data and protect the rest with locks. Minor changes was needed in several parts but also extensive rewriting of process interaction, scheduling and I/O system had to be done in some areas.
- **Scalable parallelism** - no inherit bottleneck in the runtime system should exist. Favour throughput over a maximal efficient implementation. In general, performance must not be much worse than the non-multithreaded version of today. Otherwise no one will use it.
- **Real-time** - ERLANG is not a hard real-time system and will not become one with this implementation but the design should at least have the same level of "soft" real-time performance and an embryo for further investigations into ''harder'' real time systems.
- **Dynamic** - depending on demands we like to be able to execute ERLANG processes, ports, and user defined tasks, as scheduled by workers from a run queue or run in private threads. An ERLANG process should be able to migrate from being scheduled to run in a private thread, and vice versa.

  Therefore an abstraction for any schedule activity task in the system in needed, ERLANG processes, ports and tasks. This abstraction needs to work in the multi-threaded runtime systems that mixes threads and scheduled tasks, and in a non-threaded system.
- **Portable**- Portability is important. At the start-up time of this thesis work the Posix thread standard was not yet finished so portability between Posix and thread libraries, not departing too much from the proposed standard, was desirable. Portability to Win32 threads was also desirable, if possible without extensive extra work. Existing middlewares was regarded as removing too much control; very good control and low-level optimisation possibilities was needed.
- **Maintainable** - the multi-threaded runtime system must work on operating systems without threads. On systems with one CPU the non-threaded system can still be the best choice. Because of that the runtime system must compile to two different targets; either using system threads or using internal scheduling with minimal code differences[1] and still be effective in both cases.

As a design-base to start with the ERLANG JAM implementation was chosen. Its separate process heaps with copying message passing between processes and a simple code execution model made it ideal. The unified heap implementation would had needed a thoroughly rewritten parallel and scalable garbage collector, it would also created much bigger problems with memory contention and trashing with a large number processors. The JAM machine is also the most widely used and best known implementation.

1. less code to maintain

5.2 Implementation description[1]

5.2.1 Process interaction

Processes, ports and user-defined tasks interact when they send messages, link/unlink etc. We want the synchronisation phase to be very short. The ERLANG language is highly asynchronous so it makes sense to keep the underlying implementation that way too. Therefore, all interaction is done through *signals* that encapsulates the desired functionality.

- **signals -** all processes have a signal FIFO queue that they check for new signals regularly, if there is signals a *HandleSignals()* routine is called that perform the semantics called for by the signals. A sender only need to check if the receiver exists and then insert the signal into the queue.
- **messages -** are embedded in signals, the *HandleSignal()* call moves ERLANG messages to an internal ERLANG message queue. ERLANG messages can be of any size and is part of the ERLANG heap. One exception is binaries that are an untyped area of memory, garbage collected with reference counting.

  When a process sends a message it either mallocs or fix-allocs[2] space for it on the C heap, and copies it there. The created signal has a field that points to the message and it is inserted into the receivers signal queue. If the receiver is not running it is either inserted into a run queue or a thread library notification call is performed. The receiver also has a list of temporary heaps that point to the messages, these heaps are deallocated after the garbage collector has been invoked since the living data in the messages has been moved to a new common heap.

  In the previous algorithm, the sender performed most of the message passing work. It checked if the receiver had place on the heap and if it didn't, it garbage collected it and placed the message in that heap, then it inserted the receiver in the run-queue if it was in a waiting state. The new algorithm removes some unnecessary garbage collection and also minimises the actual interaction time between processes; the sender takes a lock on the receiver for a short period of time and inserts the signal.[3]

  — *At first, another method of passing messages was implemented. It used message handles instead. The signal concept was the same but instead of the message buffer a pointer to a handle pointing into the senders heap was passed to the receiver. The sender updates the handle pointer when it performs garbage collections. When the receiver tries to match the message it fetches it from the sender and places it in its own heap. This approach worked and all different special cases were handled with dying processes etc. But the performance was worse than the original heap- to-heap message passing, and much worse than the new heap to temporary heap message passing. The performance problem probably mainly came from the many small handle allocations and the more*

1. Please note that the actual locking architecture is not included in the pseudo code samples.
2. Depending on the size, large messages uses are allocated with *malloc(),* smaller with one of several fix allocators with different sizes.
3. This message passing algorithm was quite fast adopted from this prototype and implemented into the official release of ERLANG.

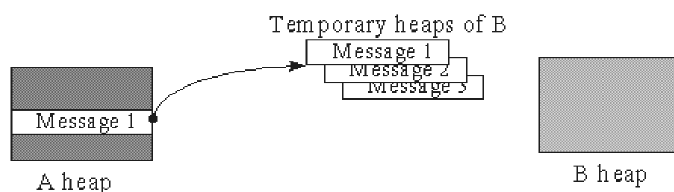
complex GC for updating the handles. An interesting extension of this method could be to make a new pattern-matching algorithm that only copy the data bound to variables at the receiver.

Figure 1. Shows how one process sends a message to another process by allocating global heap-space and copies the message to this space. The message is inserted in the receivers signal-queue. When the receiver handles the signal it sees that it is a message and inserts it in its message-queue, at the same time it also inserted as a member of the temporary heap list. After the receiver has performed garbage-collection the temporary heaps are deallocated since there are no references left to them.

Figure 1.

Messages and Signals

Process A sends a message to Process B



A places a copy of the message in global heap memory and passes a pointer to B. The message becomes a temporary heap of B and disappear after B GCs.

- **scheduling** - a sender of a signal only needs to create the signal and make a call *deliver_signal(ReceiverPid, Signal).* This function performs all the necessary scheduling, either inserts the receiver in some run-queue or notifies it with some thread notification call. In each process, port or user defined task, there are tags identifying the actual scheduling method applied for in each schedulable entity.

5.2.2 Memory management

The previous runtime system was quite static in its use of the C heap and could use the OS process stack extensively since some parts of the runtime system were recursive. I/O could be handled in a static scratch buffer. A multi-threaded runtime system has to be more careful with the stack usage since each thread has a separate[1] stack that is of fixed size and there can be many threads. So stack usage must be limited and the maximum size known.

- **stack usage -** since several threads with separate fixed-size stacks can execute the stack usage per thread must be low for space reasons and must have a known maximal size because they cannot grow. All recursive runtime code was rewritten to use explicit stack handling. The stacks can now be limited to a couple of hundred machine-words.

1. Not to be confused with the ERLANG stacks that are handled separately.

- **fix allocator** - since more dynamic memory is used in message passing and stack handling the risk for memory fragmentation increases. Since ERLANG is used to build robust systems that has to function for long periods of time fragmentation is minimised by using a fix-allocator wherever possible. *malloc()* is only used for large messages and some types of I/O.

5.2.3 Tasks

By introducing the notation of tasks, both ports, process and user defined tasks may be described by a common structure:

```
Task = RECORD
   pq:    DLink;        (* Run queue *)
   sq:    SleepLink;    (* Timeout chain *)
   id:    TaskId;       (* Task identifier *)
   prio:  INTEGER;
   links: LinkPtr;      (* Link chain *)
   sig:   SignalQueue;
   msg:   MessageQueue;
   state: TaskState;    (* current state *)
   flags: LONGINT;
   fs:    TaskDefinitionPtr;
END
```

The behaviour of a task is captured by the TaskDefinition record that defines a call-back interface:

```
TaskDefinition = RECORD
   type:      String;
   prio:      INTEGER;
   flags:     LONGINT;
   Allocate:  PROCEDURE;
   Free:      PROCEDURE;
   Init:      PROCEDURE;
   Terminate:PROCEDURE;
   Setup:     PROCEDURE;
   Idle:      PROCEDURE;
   HandleEvent:     PROCEDURE;
   HandleTimeOut:   PROCEDURE;
   HandleInfo:      PROCEDURE;
END
```

A new task type is introduced by extending Task as follows:

```
XmplTask = RECORD IS A Task OF
   counter: INTEGER;
END
```

And by specifying the following functions:

```
XmplDef: TaskDefinition := {
   "xmpltask",         (* task type name *)
   10,                 (* priority *)
   TASK_START_THREAD,  (* type flags *)
   XmplAllocate,
   XmplFree,
   XmplInit,
   XmplTerminate,
   XmplSetup,
   XmplIdle,
   XmplHandleEvent,
   XmplHandleTimeout,
   XmplHandleInfo };
```

The first entry in the task definition is the task type name, "xmpltask" in this example. Then comes task priority in range 0 to 31, where 0 is the highest and 31 the lowest priority. We also allow for the task to have some initial conditions:

- `TASK_PORT_TYPE` **-** The task will be given a port identifier, if this flag is not set the task will be given a process identifier.
- `TASK_START_THREAD` **-** Task will use a private thread at start. If this flag is not set the task will start as a scheduled entity.
- `TASK_CANT_SCHEDULE` **-** The task can not run without thread support
- `TASK_CANT_LOWER_PRIO` **-** The task can not lower its priority.
- `TASK_CANT_RAISE_PRIO` **-** The task can not raise its priority.

The call-back interface:

- **Allocate -** allocate the memory used by the task structure.
- **Free -** free the memory allocated by allocate.
- **Init -** when the task is assigned an Id then init is called, where task specific data is initialized.
- **Terminate -** called when the task has received an exit signal, then it is up to the terminate function to clean up.
- **Setup -**when the task is visible for other tasks the setup is called, i.e. other tasks may signals and messages to it.
- **Idle -** Since the task system is an event based one, idle is called when the task is scheduled but has no events. This function is where tasks may do batch job or execute sequential code. ERLANG processes do most of their work here.
- **HandleEvent -** When a task is waiting for a message and there is one in the message queue HandleEvent is called.
- **HandleTimeout -** ordering of timeouts is done by the function SetTimer. When the time has passed and the task is waiting and has no input messages then HandleTimeout is called. Note that HandleEvent has precedence over HandleTimeout.
- **HandleInfo -** Return information about the task. HandleInfo is called in response to the info signal.

From ERLANG a call to the built-in function **`spawn_task(xmpltask, Arg)`** will create an instance of the xmpltask. ERLANG processes are created with **`spawn`** and ports with **`open_port,`** but the implementation of spawn and open_port uses the same mechanism as spawn_task. In the runtime system there is a function *SpawnTask* declared as:

```
FUNCTION SpawnTask(creator: Task,
          def:   TaskDefinition,
          arg:   Ptr,
          linkit:BOOLEAN): Id
```

SpawnTask will create and initialize the task, and then schedule the task in the state returned by the *Init* function.

5.2.4 Scheduler

The scheduler is the main loop in the runtime system:

```
PROCEDURE Schedule()
VAR
   task: Task;
BEGIN
   reductions := 0;
   WHILE reductions < INPUT_REDUCTIONS DO
   BEGIN
       task := DequeueTask();
       IF task = NIL THEN
          RETURN
          RunTask(task);
   END
END Schedule;
```

The procedure DequeueTask selects the task with the highest priority and removes it from the schedule queue. The RunTask is called to perform the requires actions on the selected task. The scheduler will keep on doing this until the global counter **`reductions`** reaches a maximum value or there are no more tasks to run. The caller to Schedule will check for I/O and call schedule whenever there are more jobs to do.

```
PROCEDURE RunTask(task: Task)
VAR
   state : TaskState;
BEGIN
   state := HandleSignals(task);

   IF state = SETUP THEN
       state := CallSetup(task);

   IF state = WAIT THEN
       IF HasEvent(task) THEN
          state := CallHandleEvent(task);
       ELSE IF HasNoSignals(task) AND
            HasTimeout(task) THEN
          state := CallHandleTimeout(task)
   END

   IF state = IDLE THEN state := CallIdle(task);
   IF state = EXIT THEN CallTerminate(task)
   IF state = IDLE OR HasPendingSignals(task)THEN
       EnQueueTask(task);
       task.state = state;
   END RunTask;
```

The function HandleSignals handles general signals like **`LINK,EXIT,SUSPEND, RESUME,INFO`** etc. The state of the task is changed according to the signal. One important signal is the message that is sent as a signal and then moved to the message queue. When a sender delivers an exit signal it does not know and care if the receiver is trapping exits or not, this is handled by the receiver.

Notice that by using this signal concept much of the intricate process interaction code can be avoided and it is relatively easy to add new signals. The signal concept is also quite general, some C code was removed and replaced with internal[1] ERLANG code sending the signals and handling of them.

5.2.5 Threads

By running multiple instances of Schedule in parallel, by means of threads, we get a parallel runtime system. At start-up of the runtime system one can set the default one for new processes. The procedures DequeueTask and EnqueueTask will lock the runqueue while we examine them. DequeueTask will go to sleep if there is no more tasks to run, and EnqueueTask will wake up the schedule worker by sending it a thread notification signal.

```
PROCEDURE Schedule'()
VAR    task : Task;
BEGIN
   WHILE TRUE DO
   IF task_queue <> NULL THEN
       BEGIN
          task := DequeueTask();
          RunTask(task);
       END
   WAIT_FOR_THREAD_NOTIFICATION();
END Schedule';
```

We also have the option of having some or all tasks running on a single thread, this is achieved by a slight modification of RunTask.

```
PROCEDURE RunTask'(task: Task)
VAR state : Task
BEGIN
   WHILE TRUE DO
       state := HandleSignals(task);
       IF state = SETUP THEN state:= CallSetup(task);
       IF state = WAIT THEN
          IF HasEvent(task) THEN
             state := CallHandleEvent(task);
          ELSE IF HasNoSignals(task) AND
               HasTimeOut(task) THEN
             state := CallHandleTimeout(task)
       END
       IF state = IDLE THEN
          state := CallHandleTimeout(task)
       IF state = EXIT THEN
       BEGIN
          CallTerminate(task)
          TerminateThread();
       END
       task.state := state;

   END
END RunTask';
```

In this version of RunTask we have to modify HandleSignals to wait for a new signal or a timeout in the `WAIT` state. A sender, either a task with a a private thread or a scheduled task will wake it up with a thread-signal.

1. the signal BIF is not available for application programmers

5.2.6 Task Priority

The priority system of this new runtime system is modified from the previous one in such a way that each priority level pre-empts a lower priority level. Within a priority level the tasks are scheduled using a FIFO queue. A task may however, spend as much time as it wants in its call-back functions and it is up to the implementation of tasks not to spend more time there that acceptable.

A task executing an ERLANG abstract machine has the advantage of running emulated instructions, so a simple counter is used to control the approximate amount of time to spend within a task procedure.

The number of priority levels (32) can be justified by observing that we may need different priorities for different tasks like I/O or the ERLANG systems built-in feature of tracing messages etc, and the tasks used for I/O and processing tracing information must be able to process their work.

An example of priority defaults could be :

- 0 - reserved for system use
- 1 - reserved for tracer
- 10 - default port I/O
- 15 - high process priority
- 20 - normal process priority
- 30 - normal low priority

Processes not scheduled by worker threads but running with private threads do not follow this priority scheme; their behaviour depends on the systems thread library scheduling policy. But they still count instructions and can yield to other threads when their instruction quanta is used. A tasks that do not execute the virtual machine, but compute bound code, can hinder all other threads to execute if the the thread library cannot provide preemption at any point.[1]

5.2.7 I/O and timer

Timeouts was in the previous runtime system handled by setting a timeout value in the scheduler *check_io()*[2] call that was called at regular intervals. This is still done when the runtime system runs without threads. With threads, timeouts either are handled in a thread library timeout call if the task runs with a private thread, or if it is scheduled, by setting a timeout in a special timeout thread that handles all timeouts for scheduled tasks.

I/O is also handled in the *check_io()* call when the system run without threads, there non-blocking reads and writes are scheduled in a way that fits the ERLANG port concept. With threads each ERLANG port is a thread that controls one or more threads making blocking reads or writes.

1. like the solaris threads implementation with the defualt single "LWP", then more LPWs must be added with a *set_concurrency()* call.
2. that on Unix like systems is a select() call

5.2.8 Tasks and threads

By using threads we gain parallelism when the underlying operating system and hardware supports it, each thread run a task scheduler or a separate task. Wirth [21] discuss them as opposed paradigms but here they are shown to cooperate.

On a multiprocessor machine there can be several threads scheduling ERLANG processes, we call them workers as they can be seen as a pool of workers that you pass work to. To somewhat reduce unnecessary context-switches between these workers when they execute ERLANG processes that pass messages, but are synchronous and can not run in parallel, some optimisations can be done with what we call a ''worker-cache''. When a signal is sent from a scheduled ERLANG process to another scheduled ERLANG process that is not running, and the run-queue is empty, the receiver process is inserted in the worker thread cache that executes the sender, if the cache is empty. This can keep a a large amount of sequential ERLANG process interaction within one worker and unnecessary thread context switch can be avoided:

```
PROCEDURE Schedule''(worker: INTEGER)
VAR    task : Task;
BEGIN
    WHILE TRUE DO
    BEGIN
        IF worker_cache[worker] <> NULL THEN
        BEGIN
            task = worker_cache[worker];
            worker_cache[worker] = NULL;

            (* prio check can be omitted if not strict
               priority fairness is needed, more even effective
               since less locking is needed *)
            IF task->prio >= HIGHEST_IN_RUN_QUEUE() THEN
               BEGIN
                  ENQUEUE_TASK(task);
                  CONTINUE;
               END;
            task->worker = worker;
            RunTask(task);
            CONTINUE;
        END
        IF task_queue <> NULL THEN
        BEGIN
            task := DequeueTask();
            task->worker = worker;
            RunTask(task);
        END
        (* worker waits here for new work in run_queue *)
        WAIT_FOR_THREAD_NOTIFICATION();
    END;
END Schedule'';
```

An ERLANG process can change from being scheduled by a worker or run on its own thread by a special signal. The two scheduling mechanisms has different advantages. It is in some cases safer to let a process execute on a private thread, for example, if it performs calls to native linked in code that can take a ''long'' time to return from.

Some ERLANG systems has many thousands of ERLANG processes, if the target OS has threads that are relatively heavy-weight the worker-threads must be used for a majority of the ERLANG processes or the system may become to large and slow.

The worker threads also has the advantage of giving the scheduled processes an platform independent priority and schduling mechanism.

Figure 2. The mixed scheduling figure shows how the different concurrent entities interact in the runtime-system when it runs. The figure shows a system with a total of nine

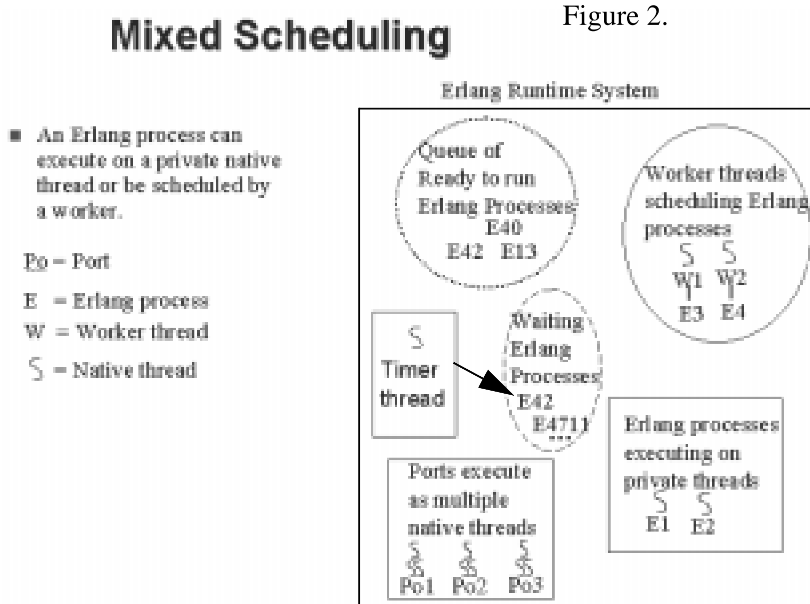


Figure 2.

ERLANG processes. Two schedule processes are at this moment executing on the two worker threads, two other ERLANG processes has private threads and can be in any state. Two schedulable processes are waiting, one for a message from a port or processes. The other is also waiting for a message but has also set a timeout that can be triggered by the timer thread, this is indicated by the arrow. Three schedulable processes is ready to execute in the run-queue, when the processes executed by the workers has run out of instruction reduction quanta, or starts to wait for a message, or timeout, the workers picks new processes to execute from the run-queue. The system has three ports; each port has one controller thread and one reader and one writer thread each.

5.2.9 Dyncamic code handling

It is up to the application programmer to check that no process runs old code via the **`check_process_code()`** call before deleting a module. A deleted module is not really deleted, only the exported references is deleted so no processes can start execute code in it and the module is marked a old. Finally a **`purge_module()`** call deletes the module from memory, then no process may execute code in that module.

Throughput is not important in this procedure since code swap and loading is uncommon. Calls to **`load_module()`** are serialized and the new module is inserted in the running system with one atomic insert of a new module pointer to this module. It is important to realize that no process can start execute code in an module marked as old. So after **`delete_module()`** is called no new references to this module can appear. Functions trying to call a non loaded mod-

ule gets an error and the error handler then tries to call **`load_module()`** where it either discovers new code, returns and continues to execute; or loads the code, returns and continues to execute; or, if there is no such code available returns an error.

This approach was later replaced with a new module implementation with a changed module semantics. That implementation solves the dynamic code handling problem. The new implementation is not described in this thesis.

6 Measurements

A number of different micro-benchmarks have been performed on the new multithreaded ERLANG implementation, they can be divided into two different categories; verify the actual parallelism of the implementation on an MP machine[1], and to compare the performance with and without threads when running selected parts of the EStone[2] benchmark.

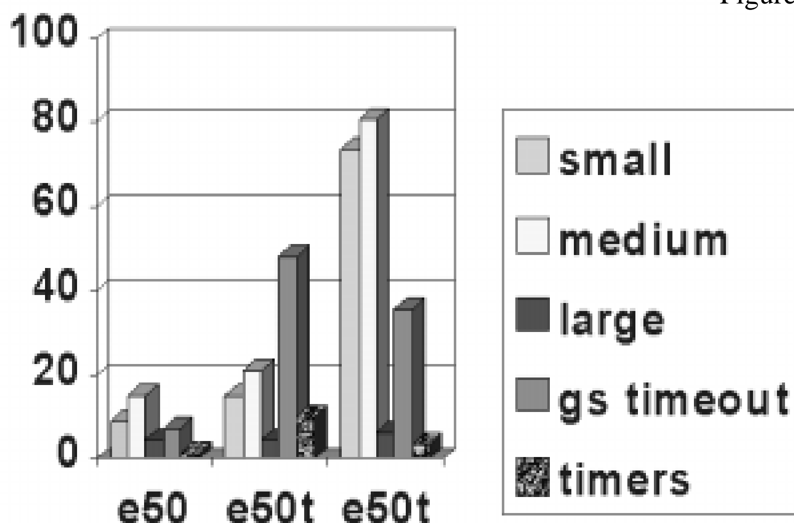


Figure 3. Shows the result of running five parts of the Estone benchmark suite on three different configurations of the multithreaded ERLANG runtime system.

— *e50* is without any threads at all; the basic scheduling is as in the old system[3].

— *e50t* is with worker threads as default when a new process is started; no processes with private threads exist in this run.

— *e50 +T* is with private thread per process as default when a new process is started, no processes is scheduled with workers in this run.

The bars are relative and should only be compared between the different implementations. The height of the bar indicates time.

Message passing is performed between a group of processes in the three first runs.

— *small* consists of one small integer.

— *medium* a mix of datatypes with a size of about a line in this thesis.

1. All tests were performed on an Compaq ProLiant 5000 with four 200MHz PentiumPro processor with 512KB of 2:nd level cache and 64MB of RAM running Solaris 2.5. The ''concurrency'' level in the Unix process was set to be the number of processors plus one, five on this system. All threads are ''unbound'' from LWPs.
2. A ''standard'' micro-benchmark for ERLANG to compare different ERLANG implementations.
3. The performance were very simmilar between them when the old system had been updated with the temporary heap method of message passing.

— l*arge* is about half a KB in size.

Note that it is not necessarily the same number of messages for the different message sizes.

This benchmark mainly shows two things, message passing and context-switch cost. The message passing cost grows with the size. One can see that the difference between the *e50* and *e50t* message passing runs are minor. The *e50t* has slightly higher context-switch cost; for large messages the difference is very minor for all three runs since the cost of context-switches is small compared with the cost of copying the actual message. The *e50t +T* runs has substantially higher values for small and medium message. The conclusion is that the context-switch cost is substantially higher when ERLANG processes run with private threads and thread synchronisation primitives are used between them than the one without any threads using internal scheduling and the one with worker threads.

The large messages gives little difference between the runs since the cost of copying the messages are so high compared with the context-switch cost.

The *gs timeout* testrun is an EStone benchmark using the *generic_server* behaviour from one of the ERLANG libraries. It implements a server process and clients processes using the server, the clients are synchronous so no parallelism is possible but timeouts are set and removed before they are triggered at each request. This benchmark tests a mix of message passing and context-switches together with setting timeouts. The timeouts are never triggered. Here it shows that the one using worker threads is the worst performer.

In the *timeout* benchmark a process is setting and resetting timers that never triggers. In this test it is obvious that the worker thread implementation with a timeout thread is slower than the two others, the best timeout implementation can be found

The system running without threads has the best timeout implementation, the one using a thread for each process with a thread library timer timeout call is a bit slower and the worker thread processes using the dedicated timeout thread is the worst performer.

Figure 4. Here the time for a simple computing program to finish its calculation is measured, the time is measured for one process executing the program and for two and so on up to six identical parallel processes. Without parallelism the time should increases with a fairly constant factor for each new process. With parallelism the time should be fairly constant for each new process for as long as there is processors. The graph shows this clearly for the different runs, *erl50* that is the runtime system without threads and *erl50_thr* that is the runtime system with threads[1]. There are four processors on the machine and the runtime system with threads can use them. The system without threads cannot use the extra processes.

The test is very simple but shows that parallel hardware is used efficiently.

1. worker threads only, but a private thread test were also done with minimal difference.

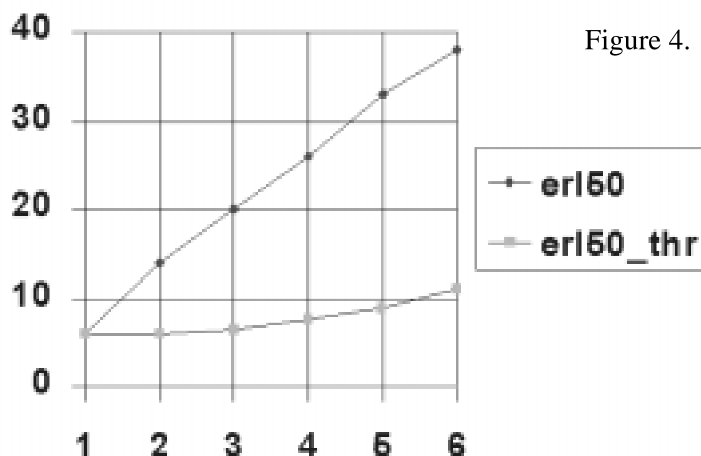


Figure 4.

In these tests one has to remember that for the worker thread and in particular, the thread per processor, test values can vary a great deal depending of the performance of the thread library, of the underlying platform.

Several other micro-benchmarks programs were also run that, at large, shows the same results as the above tests:

- That parallelism is utilized, but that it is highly dependent on the application; the amount of process communication needed and the message size.
- That the worker thread implementation has much lower context-switch cost than the one using a thread[1] per ERLANG process. Actually the context-switch cost is only marginally higher than without threads for the worker thread model.
- The timeout implementation for worker-threads has bad performance.

  Now a process takes a lock on the timer thread and inserts a timeout in a timer wheel [22] and then releases the lock and notifies the timer thread that something has changed. Maybe a special in-queue can be used where new timeouts are inserted, and the timer thread is only notified of new timeouts if the new timeout needs to be triggered sooner than the one the timer thread is waiting for now. This would minimize thread synchronization and time wheel computations since many removals of timeouts could be done in the in-queue before the timeout is inserted into the time wheel.

Other benchmarks also gave results that I interpreted as:

- The worker-cache optimisation improved the context-switch performance with ~5-15%.
- The I/O model for the system using threads has very bad performance compared with the one without threads. Probably because extensive context-switching between ERLANG processes and port threads.
- The added cost at runtime when merging a worker based scheduler with a thread per process based system into a mixed scheduling system was very minor.

1. Again, this is for Solaris 2.r threads, this can change a great deal for other systems, both better or worse.

- Different locking implementations were tested at different places in the runtime system. For the low-level db routines that handles many internal tables in the runtime system and is used by the ETS[1] module for its implementation, three different lock implementations were tried. The native thread library mutual exclusive locking, reader-writer locks allowing several readers but exclusive writing were built with the thread library locks and condition signals, and a low level assembler[2] mutual excluse lock implementation. The results were hard to interpret; the reader-writer locks were generally slower, a large number of processes must try to read and write the same table before any performance gain could be seen at all. The assembler locks were slightly faster than the thread library locks but when many processes were involved in the tables the scheduling behaviour became unpredictable. The native thread locks are used as default when compiling the system now but the two others can be used instead by simply changing compile configuration flags.

1. ERLANG Term Storage
2. Using the processors built-in instructions to build multi-processor locks.

7 Conclusion

A runtime system has been created that can be compiled in two ways; a *select()* based system using reduction counting scheduling and a system using threads. When threads is used the ERLANG processes can either be part of a run-queue that a pool of workers picks them from to execute or the processes can execute on private threads.

This mixed scheduling does not add any extra runtime cost and gives greater flexibility on how to setup an multi-threaded ERLANG system. On Solaris 2.5 the worker approach gave better context-switch performance compared with the thread per process. Potential advantages from the private thread model can still be used; like improved real-time characteristics and integration with C code with less risk of corrupting the scheduling of other ERLANG processes. A process running on a private thread can migrate to become a worker thread scheduled process and vice-versa with a signal. A ''worker-cache'' is implemented for workers scheduling ERLANG processes, this further improved context-switch perfromance.

When the system is compiled not to use threads the extra runtime cost for making the system general enough for the different compiles is very minor compared with the previous system. The actual code needed to support the two compiles is minor.

When the hardware and the thread library provides parallelism the runtime system also provides it in the same degree. The system is designed to be parallel and scalable, tests showed it to be so for four processes. The actual level of parallelism varies depending on the application. One can assume that many applications would benefit from better load tolerance and less often through actual parallel problem solving speed-up

The result from micro-benchmarks are fairly positive, but the I/O system must be redesigned to be more effective. Efforts has been done to keep the system portable between Posix systems and Posix ''like'' systems. But it is definitely not portable to Win32 now, mainly the I/O system needs to be reworked for portability and speed.

Timeouts for workers scheduled processes is slow and has to be improved - one proposal is made how to improve this.

So far, only micro-benchmarks has been run, no ''real'' applications.

Most of the goals have been fulfilled with this implementation. At large, a parallel and multi-threaded ERLANG implementation can be efficent, portable, scalable and keep or improve the original benefits of the previous implementation.

References

[1] Joe Armstrong, Robert Virding, Mike Williams, Concurrent Programming in ERLANG, Prentice Hall, 1993.

[2] A comprehensive article describing ERLANG System/OTP: www.ericsson.se/Review/er1_97/art_2/art2.html

[3] Some products where ERLANG is used in: www.erlang.se/erlang/sure/main/applications/

[4] Brian W.Kernighan, Dennis M Ritchie, The C Programming Language, Second Edition, Prentice Hall, 1988.

[5] Virding, S. R. Garbage Collector for the concurrent real-time language ERLANG. International Workshop on Memory Management (IWMM '95) September 27-29, Kinross, Scotland.

[6] Hausman, B. Turbo Erlang: Approaching the speed of C. In implementations of Logic Programming Systems. Evan Tick, editor. Kluwer Academic Publishersm 1994.

[7] Overview of Scalable Shared Memory Multiprocessor Principles http://www.sics.se/cna/mp_overview.html

[8] Simple COMA on S3.mp by A Saulsbury, A Nowatzyk in http://www.sics.se/~ans/simple-coma/index.html.

[9] The SGI Origin: A ccNUMA Highly Scalable Server. www.sgi.com/Technonlogy/Compcon/icsa.html.

[10] SUN Enterprise X000 Server Family: Architecture and implementation. www.sun.com/servers/enterprise/arch.html.

[11] Curt Schimmel, ''UNIX Systems for modern Architectures - Symmetric Multiprocessing and Caching for Kernel Programmers'', Addison-Wesley, ISBN 0-201-63338-8.

[12] John L Hennessy & David A Patterson, ''Computer Architecture A Quantitative Approach'', Morgan Kaufmann Publishing, ISBN 1-55860-329-8.

[13] Bil Lewis, Daniel J Berg, ''Threads Primer'', Prentice Hall, ISBN 0-13-443698-9

[14] Steve Kleiman, Devang Shah, Bart, Bart Smaalders, ''Programming With Threads'', Prentice Hall, ISBN 0-13-172389-9

[15] David R. Butenhof, ''Programming With POSIX Threads'', Addison Wesley, ISBN 0-201-63392-2

[16] Jim Beveridge, Robert Wiener, ''Multithreading Applications in Win32'', Addison Wesley, ISBN 0-201-44234-5

[17] Andrew D. Birrel, ''An Introduction to Programming with Threads'', Digital System Research Center, 1989.

[18] ACE - An OO Encapsulation of Lightweigth OS Concurrency Mechanisms in the ACE toolkit. www.CS.wustl.edu/~schmidt/ACE-concurrency.ps.gz

[19] A Library Implementation of POSIX Threads under UNIX, by F MUeller, Win-

ter USEIX - January 25-29, 1993 - San Diego, CA

[20] SunOS Multi-thread Architecture, M L Powell, S E Kleinman, S Barton, D Shah, D Stein, M Weeks. USENIX Winter '91 - Dallas TX.Wirth, N. Tasks verus Threads: An alternative Multiprocessing Paradigm. Software concepts and tools (1996): 17 6-12, Springer Verlag 1996.

[21] Wirth, N. Tasks versus Threads: An Alternative Multiprocessing Paradigm. Software Concepts and Tools (1996) 17: 6-12, Springer-Verlag 1996.

[22] Hashed and Hierarchical Timing Wheels. Efficient Data Structures for Implementing Timer Facilities. George Varghese and Tony Lauck.