Evaluation of HiPE, an ERLANG Native Code Compiler.

Erik Johansson
Computing Science Department,
Uppsala University
Box 311, S-751 05 Uppsala, Sweden
happi@csd.uu.se

Sven-Olof Nyström
Computing Science Department,
Uppsala University
Box 311, S-751 05 Uppsala, Sweden
svenolof@csd.uu.se

Thomas Lindgren
Bluetail AB
Hantverkargatan 78
S-11238 Stockholm, Sweden
thomasl@csd.uu.se

Christer Jonsson
Apicula
Hantverkargatan 40
S-112 21 Stockholm, Sweden
Christer.Jonsson@apicula.se

October 12, 1999

Abstract

In the telecom industry the software requirements often include the need to upgrade the system at run-time, and at the same time the development has to be quick in order to keep the time to market short.

ERLANG/OTP addresses many of the problems inherent in software for the telecom industry. The ERLANG systems of today are slow, even though they are "fast enough" for most applications.

Still there are some time critical applications that could take advantage of a faster ERLANG implementation. At the **Hi**gh **P**erformance **E**RLANG (HiPE) group at Uppsala University we work with the execution behavior of ERLANG. We intend to find out how to design a compiler for large, industrial, functional programs.

In this report we will describe and compare three run-time systems for ERLANG: JAM, BEAM, and HiPE. We will describe some hardware aspects of a modern RISC processor: the UltraSparc™. We will also describe some techniques for performance measuring: both some hardware specific and some ERLANG specific techniques.

These techniques will then be used to thoroughly examine three ERLANG programs: parts of two large telecom applications and one small sequential benchmark. From the collected data we can examine the effects of the cached memory hierarchy, prediction, and pipelining on the three run-time systems.

We will show that our system (HiPE) is 1.6 times faster than the emulated byte code system JAM, provided by Ericsson, on a large time critical industrial benchmark, even though 32 percent of the time is spent in built-in functions, which currently is outside our control.

We will also show that even though large programs, when they are compiled to native machine code, have trouble with the instruction cache this problem is not larger than the pipeline problems emulated code has, such as misprediction stalls.

Contents

(This page intentionally left blank.)

1 Introduction

Telephone switches have to be very reliable (the down times are measured in minutes per year, and the uptime should be at least 99.998%), therefore there are high demands on the software. In the telecom industry the competition is hard and the technological evolution is fast. This leads to the need to quickly develop robust applications that easily can be maintained and extended.

The concurrent functional programming language ERLANG is designed by Ericsson with the needs of the telecom industry in mind. ERLANG addresses these needs with a run-time system that provides many features often associated with an operating system, such as scheduling of concurrent processes, memory management and networking. With the Open Telecom Platform (OTP) ERLANG is further extended with a library of standard solutions (servers, state machines, process monitors, load balancing), standard interfaces (CORBA), and standard communication protocols (http, ftp). The ERLANG implementations of today are slow, but since ERLANG addresses the problems in the telecom industry it has been very successful.

We, the HiPE group, intend to develop a high performance ERLANG implementation that compiles ERLANG to native code. We have begun this work by implementing a simple ERLANG to SPARC compiler. This allows us to do precise measurements on real ERLANG programs in order to find bottlenecks to concentrate our efforts on. Typical telecom applications are large and have inner loops that are too large to fit in the instruction cache. They also involve process communication and process scheduling. It is not obvious that traditional optimization techniques will be effective on these applications.

One aspect we wanted to investigate was whether native code would have serious problems with the instruction cache on large programs.

In this paper we will present our findings when examining two time critical telecom applications, namely parts of Ericsson's AXD 301 ATM switch, and the web server Eddie.

We have run these benchmarks in three ERLANG run-time systems that have much in common, such as the same garbage collector and built-in functions.Still they differ in some important aspects, two of the systems uses different types of abstract machines and one system compiles to native code. This makes it possible for us to compare these designs to each other.

We have done our comparison by measuring the low level aspects of the execution behavior, such as the time the CPU spends stalling because of different types of cache misses.

When we compared emulated and native code we found that the total amount of pipeline stalls is larger for the emulated code than for the native code, even when the native compiled code is so large that the inner loop does not fit in the instruction cache. We suspect that three factors are responsible for this: The emulated code has to use the same data cache as the application data. The stack-based emulator uses more load and store instructions than a register implementation. The branching behavior of the general byte code emulator is harder for the hardware to predict than that of specialized native code.

We also found that a large portion of the execution time of this application is spent in built-in functions.

The paper begins with a brief presentation of ERLANG (Section 2)[1].

Then we will describe three different ERLANG run-time systems JAM, BEAM, and HiPE (Section 3). We will then describe the experimental setup that we have used (Section 4).

After that we will show an encouraging result on a small toy benchmark (Section 5) and on a part of the web server Eddie (Section 6). Then we will present the AXD 301 benchmark in some detail with results (Section 7). Finally we will present some ideas about future work (Section 8) and our conclusions (Section 9).

[1]If you are interested in a more complete description of the language please look at the free ERLANG site `www.erlang.org`, Ericsson's commercial ERLANG site `www.erlang.se` or read Concurrent Programming in ERLANG [2].

2 Erlang

ERLANG is a concurrent functional programming language. The run-time system of an ERLANG implementation has many features more commonly associated with operating systems: concurrent processes, scheduling, memory management, distribution, networking, etc.

2.1 Basic properties of Erlang

There are no destructive updates in ERLANG. In the example (Example 1), the variables `Rest`, `AccLen`, and `Length` are immutable as all ERLANG variables.

Example 1 (Two simple functions in Erlang)

```
% Comments are preceded by a percent sign...
%  ...and run to the end of the line.

% The function len/1 calculates the length of a list.
len(List) ->
 len(List, 0). % By calling len/2 with the list and zero.

% The function len/2 calculates the length of a list.
len([_|Rest], AccLen) ->
 len(Rest, AccLen+1);

len([],Length) ->
 Length.
```

ERLANG has no iteration constructs, but loops can be constructed by recursion. Preferably by *tail-recursion*: if the last instruction in a function is a call then that call is a tail-call. Before a tail call the current stack frame can be freed, making it possible to execute loops in constant stack space, this is called tail call optimization or last call optimization. In the example the recursive call to the function `len/2` is tail-recursive.

ERLANG is dynamically typed and there is no explicit way for the programmer to specify new datatypes[2]. But there are implicit ways to construct complex data structures from the datatypes present in the language. The simplest datatype in ERLANG is the *atom*, two atoms are identical if and only if they have the same name. There are three different types of numbers in ERLANG: *fixnums*, *floats*, and arbitrary precision numbers (*bignums*). ERLANG has some other simple datatypes, such as process identifiers (*PIDs*), *references*, and *ports*. There is also a special datatype, called a *binary*, for (large) sequences of bits,

[2] ERLANG does have a definable data structured called a record, but records can be converted to tuples by a preprocessor step.

which is often used for incoming and outgoing communication. The implementation of binaries has to be both time- and space-efficient since large binaries often are used in ERLANG programs that deal with for example protocol stacks.

ERLANG offers two ways to build complex data structures. All ERLANG datatypes can be combined into polymorphic *lists* or *tuples*. A list is either empty (`[]`), called *nil*, or a *cons* of some datatype and a list (`[Any|List]`). A tuple of arity N is a vector of N elements with constant access time for each element ($\{E_1, E_2, E_3, \ldots, E_N\}$). In the example there are two constants: the empty list `[]` and the fixnum `1`.

ERLANG supports *pattern matching*, where patterns of datatype constructors can be used to distinguish between different cases. An unbound variable in a pattern matches any term at that position of the term being matched. When a match is successful the variables in the pattern are bound to the corresponding terms. The universal pattern '`_`' matches any ERLANG term.

In the example the function heads in the two clauses of `len/2` have distinct patterns. The pattern '`[_|Rest]`' matches all lists with at least one element. When a match is successful the rest of the list (all but the first element) is bound to the variable '`Rest`'. The pattern '`[]`' only matches the empty list.

2.2 Modules

A module in ERLANG is a collection of functions sharing the same name space. Functions in one module are accessible to functions in other modules only if they are explicitly exported from the module defining them. Within the module however all functions are visible. Example 2 shows a complete ERLANG module.

A call to a function in another module is called a remote call. The compiler does not need to check that functions in other modules exist. But if the destination of a remote call does not exist when called at run-time, a run-time error is generated.

Example 2 (A module in Erlang)

```
-module(length).
-export([length/1]).

% Calculates the length of the list List.
length(List) ->
  len(List, 0).

% Tail-recursive implementation that uses an
% accumulated parameter to calculate the length of a list
len([_|Rest], AccLen) ->
  len(Rest, AccLen+1);
```

```
len([], Length) ->
  Length.
```

A unique feature of ERLANG is its ability to change code in a running program, called *hot-code loading*. Old code can be phased out and replaced by new code on module at the time. During the transition, both old code and new code can coexist. When new code for a module is loaded each remote function call to that module will be to the new code. It is thus possible to install bug fixes and upgrades in a running system without disturbing its operation.

This means that there have to be mechanisms in the run-time system to facilitate code replacement. One way to do this is by using a dynamic lookup for each remote call. Another approach is to let the call include the real address of the destination and then patch each remote call site with the address of the new code. These mechanisms can both incur run-time costs and make optimizations, such as inlining, harder.

2.3 Concurrency

Concurrency, which is central to ERLANG, is achieved by independent ERLANG processes. Conceptually, processes have no shared memory, instead they communicate by asynchronous message passing. Example 3 shows how a new process is created and how a message is sent to that process.

Example 3 (Process communication in Erlang)

```
-module(process_comm).
-export([a_process/0, start/0]).

start() ->
  A_PID = spawn(process_comm, a_process,[]),
  A_PID ! {ping, self()},  % send a message.
  receive                 % Wait for a message...
    pong -> true;         % We hope to get a pong back.
    _ -> false
  end.

a_process() ->
  receive
    {ping, B_PID} ->
         B_PID ! pong;          % Receive a ping and respond
    _ ->
          throw(unknown_message) % We don't expect this message
    after 10000 ->
          throw({time_out})      % If we don't get any message...
  end.
```

The primitive `spawn/3` creates a new process and returns the process identifier (PID) of the new process. The two first arguments to `spawn` are atoms representing the module name and function name of the function that the new process should execute. The third argument is a list whose elements are passed as arguments to the function, the length of the list gives the arity of the function. Only exported functions can be given as argument to a spawn, that is why `a_process/0` is exported in the example.

A message can be sent to a process using the infix send primitive `!/2`, as in the example, where a tuple is sent to the new process with:

`A_PID ! {ping, self()}`

When a message is sent it is placed last in the ordered *mailbox* of the receiving process.

The built-in function `self/0` returns the process identifier of the current process.

Pattern matching can be used to distinguish between incoming messages in the `receive` primitive. In the simplest form the pattern is just a free variable as in:

```
receive Message -> Message end
```

This expression will check the mailbox for *any* messages and return the first message in the mailbox. If the mailbox is empty the process will be *suspended* until it receives a message. (In the general case the process will be suspended if there is no message in the message queue that matches the patterns.) If the suspended process receives a new message it checks if the message matches any of the patterns; if it does, the process starts running again with a new time-slice, otherwise it will keep waiting.

If a process receives messages that do not match any pattern the mailbox might grow, therefore it is customary to include a catch-all with the universal pattern '_' as in Example 3 to get rid of unwanted messages.

The `receive` checks the first message in the mailbox against all patterns. If no pattern matches then it checks the next message against all patterns, and so on, until all messages are tested.

When writing robust network applications one would often like to take some special action if an expected message does not arrive on time. This can be done by setting up a *timeout* in the `receive` by using the construct `after TIME ->`, as in the example (Example 3) where a timeout of 10,000 milliseconds is set.

If a suspended process has set a timeout it will be rescheduled when the given time has expired, and execution will continue in the body of the `after` clause.

From the programmer's point of view, the same message passing mechanism that is used between ERLANG processes is also used between ERLANG processes and the outside world. This mechanism is used for communication with the host operating system and for interaction with programs written in other languages.

2.4 Exceptions

Error recovery is an important part of ERLANG and all run-time errors are trapable by means of a *catch*. A catch works like a *handle* in ML and all exceptions generated in a "catched" expression will cause the program control to be transferred to the catch. The programmer can also define his own exceptions and generate them by means of a *throw*.

2.5 Meta call

ERLANG provides the ability to do a meta call with the built-in function *apply/3*. The function `apply` takes the name of a module and a function and a list of arguments just as `spawn`, but it does not create a new process; instead it calls the given function and returns the value of the application.

Only exported functions in a module can be called with `apply`, even if the call is done from inside the same module as the called function.

2.6 Memory management

There is no explicit memory management in ERLANG, instead the ERLANG run-time system is responsible for deallocating unused data. This can be done by Garbage Collection.

2.7 Distributed Erlang

ERLANG can be run in two different ways, either locally or distributed. If ERLANG is run distributed then each instance of the ERLANG run-time system is called a *node*.

Several ERLANG nodes can be connected to each other and messages can be sent between processes on different ERLANG nodes in the same way as between processes in a local ERLANG system.

2.8 Built-in functions and libraries

The power of Erlang is further enhanced by a number of powerful built-in functions (BIFs) and an extensive standard library[3].

One important set of built-in functions is the set of database functions. These databases are local to an Erlang node and are used by the *ETS* (*Erlang Term Storage*) standard library to implement a general database. The ETS module is more or less just an interface to the built-in functions but they make it possible to use databases across Erlang nodes in a transparent manner. These databases, called *ETS-tables*, can either be private to a process or public to all processes that have access to the unique reference to the table. The data in the databases are tuples where the first element in the tuple is the key.

[3]A specific telecom library provided by *OTP* is also available for Erlang. Information about OTP (and Erlang) can be found at `http://www.erlang.se`

3 Three Erlang run-time systems

We have looked at three different ERLANG implementations: JAM, BEAM, and HiPE[4]. JAM and BEAM are two abstract machines with emulators implemented in C. HiPE is a native code implementation for SPARC. All three systems have their own compiler. In this section we will describe some aspects of these implementations and the similarities and differences between these systems.

These three run-time systems are very similar in some aspects. They use the same standard libraries, they have the same scheduler, the same garbage collector and the same implementation of built-in functions. JAM and HiPE also has the same front end, pattern matcher and the same taging scheme. This makes it possible to compare how the things that makes them different affects performance. That is, the back ends and the emulators.

3.1 Common framework

All three run-time systems are based on the same run-time "kernel"; they have the same built-in functions, the same garbage collector and the same scheduling mechanism. Here we will describe how these systems handle processes and scheduling.

3.1.1 Processes

From the view of the operating system, the ERLANG run-time system is merely an application with processes represented using ordinary data structures. An ERLANG process consists of a *process control block* (PCB), a mailbox, a stack and a heap.

An ERLANG process is extremely lightweight and these run-time systems supports applications with very large numbers of concurrent processes [7].

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.

The databases, used for example by ETS, are placed outside the processes, as shown in Figure 1. This means that data has to be copied to and from the process heap and the database when a process performs a database access.

[4] We will use the names JAM, BEAM, and HiPE to refer both to the run-time system and the compiler. We might also say that a benchmark was run on JAM when we really mean that it was run as emulated JAM code in the HiPE run-time system, since the JAM emulator is (almost) the same in HiPE as in JAM.

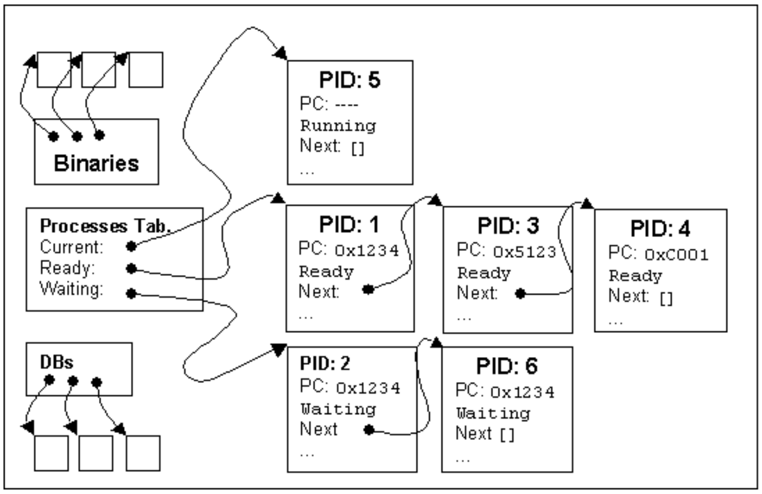


Figure 1: ERLANG processes and other data structures in the run-time system.

Binaries are also stored outside the processes, but in this case data would have had to be copied by for example the built-in function `list_to_binary/1` anyway. Since each process has its own heap this implementation technique can save space when many processes have access to the same huge binary.

3.1.2 Scheduling

The *process table* is a data structure responsible for keeping track of all processes; this is done by linking the PCBs as shown in Figure 1. Besides keeping track of the process that is currently executing, the process table contains a *ready queue* with processes awaiting execution, and a queue of processes waiting for a message or a timeout.

The top-level loop of the run-time system does two things: it checks for I/O (on sockets and file handles) and then it runs the *scheduler*. The top-level is represented by the oval in the upper left corner of Figure 2.

When the scheduler starts, it checks if a timeout has occurred for any processes; in that case those processes are placed in the ready queue.

The scheduler then selects the first ERLANG process from the ready queue. (The ready queue is really a queue in order to maintain a round-robin scheduling policy.) This process is assigned a number of *reductions* to execute. The time it takes to execute these reductions is called the *time-slice* of the process. Each time the process does a function call a reduction is used. The process is sus-

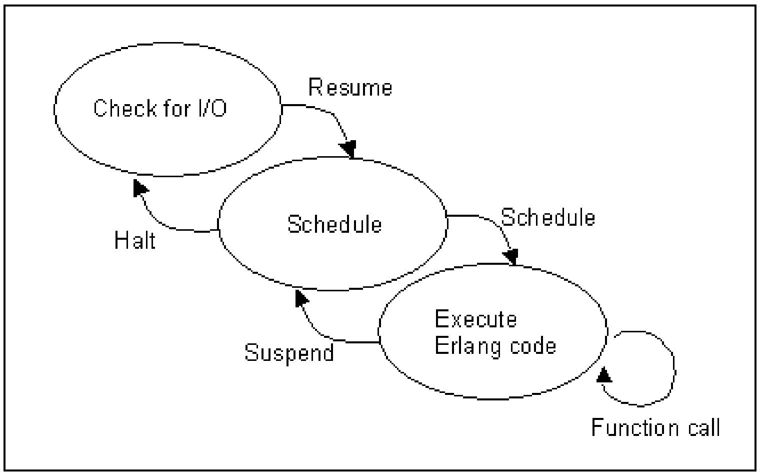


Figure 2: The execution loop of the ERLANG run-time system.

pended when the time-slice is up (the number of remaining reductions reaches zero), or when the process reaches a `receive` and there are no matching messages in the mailbox.

Each time a process gets suspended the scheduler places the process last in the ready queue, and adds the number of executed reductions to a running total. When the running total exceeds a *major time-slice* the scheduler is halted and control is returned to the top-level loop.

If the size of the major time-slice is increased, more time is spent executing ERLANG code but the interactivity of the system decreases since I/O is checked less often.

3.2 JAM

JAM [24, 3] is a stack-based byte code emulator for ERLANG. The version of JAM that we will be looking at is a modified version of JAM version 4.5.3. To this version we have added support for measurements and support for the integration with HiPE. The development of JAM has not stood still while we have been developing HiPE, the latest version of ERLANG provided by Ericsson is at the time of writing 4.8.

3.2.1 Code

The JAM instruction set is implemented with byte codes, making it compact. Code is loaded one module at the time, when needed. Code is patched at load time (for example with indices into the atom table).

3.2.2 Data representation

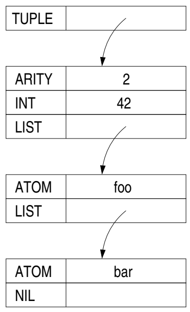


Figure 3: Representation of the term {`42, [foo, bar]`}.

All basic values are represented in one machine word (in this case 32 bits). A tag is stored in the four most significant bits, leaving 28 bits for the value. Small integers, atoms and `[]` (the empty list) can be stored directly in a machine word. For more complex values, such as lists and tuples, a pointer to a heap-allocated object is stored in the word. A list cell on the heap is just two consecutive words. Tuples consist of a header, containing the arity of the tuple, and their elements, as in Figure 3.

3.2.3 Emulator

The C-compiler *gcc* has a feature that makes it possible to take the address of a label and use it as a code pointer. This feature is used in the JAM system to implement so called threaded emulation.

3.2.4 Calls

According to the specification of ERLANG, there are two types of calls: local calls (calls within a module) and remote calls (calls to functions in other modules). Since all functions in a module are loaded at the same time the relative address for the destination of a local call can be determined at compile time. However, the destination of a remote call can change at run-time, therefore the emulator has to perform a table lookup for each remote call.

Two "registers", called ARGS and VARS, are used to keep track of the stack: ARGS points to the first argument of the current function, and VARS points to the first local variable.

Before entering a function, the function arguments are pushed on the stack (as shown at the top of the downward growing stack in Figure 4). Then a stack frame is written, containing the return address, a pointer to the code of the calling function (CC, Current Call), and the old values of ARGS and VARS. ARGS is set to point to the first argument, and VARS to point to the first free stack position. When the function returns, the frame is popped from the stack and the return value is pushed (on the same stack position as the first argument).

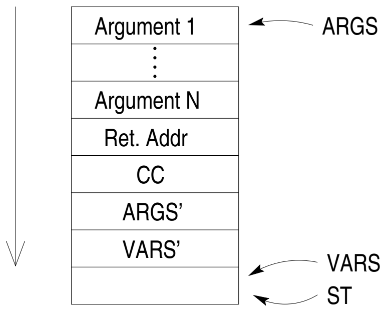


Figure 4: The stack after a call.

3.2.5 Tweaking

When we did our first measurements we discovered that the JAM system spent much more time in privileged mode than HiPE and BEAM. This was because the JAM system had a much smaller major time-slice than BEAM and HiPE. Therefore the JAM system spent more time polling for I/O.

Since there is no reason why JAM should have a higher level of interactivity than BEAM or HiPE we increased the major time-slice for the JAM system. The response time to external events (such as socket communication) will increase because of this modification, but as for now we have not noticed any ill effects because of it.

3.3 BEAM

BEAM [8, 9, 10] is a threaded emulator for an abstract register machine. The design of BEAM is influenced by the Warren Abstract Machine (WAM)[1], which was designed for the language Prolog and used in many Prolog implementations.

3.3.1 Data representation

All basic values are represented in one machine word (in this case 32 bits). Like JAM BEAM uses 4 tag bits, but unlike JAM, they are stored in the 4 least significant bits of the word. A list cell on the heap is just two consecutive words. Tuples start with a header containing the arity of the tuple, followed by their elements.

3.3.2 Emulator

The temporary and permanent registers are called X and Y registers as in WAM. These "abstract machine registers" are in reality stored in memory in an array, except for register X0 which is stored in an actual machine register.

The emulator is directly threaded; each instruction in the code is a pointer to the part in the emulator that implements the instruction. The emulator reads this instruction from memory and jumps to the code it points to. As for JAM this means that the *gcc* compiler is needed, since the address of a label in the C-code can be accessed in *gcc*.

For most instructions, the emulator first prefetches the next instruction to execute, before it starts executing the current instruction.

3.3.3 Code

Some information, such as atom numbers and function indices, is not known at compile time. Therefore the code is patched with this information at load time.

Some instructions are also replaced by specialized versions at load time. For example: instructions that take register X0 as an argument are replaced by a special version of the instruction that directly works on the X0 register.

It is also at load time that the external representation of each BEAM instruction is replaced by the actual address to the emulator code for that instruction.

3.3.4 Compilation

The BEAM compiler does a somewhat better job at optimizing the ERLANG code than JAM. For example pattern matching is compiled better than in JAM, even though a full pattern matching compiler as described in [19] is not implemented.

3.4 HiPE

HiPE is the name of our project, the **Hi**gh **P**erformance **E**RLANG project at Uppsala University. We have developed our run-time system on top of version 4.5.3 of Ericsson's JAM[13]. The main goal has been to implement a native code compiler for the SPARC architecture.

3.4.1 Integration with JAM

In the HiPE run-time system we can run both emulated JAM code and native compiled SPARC code, within the same ERLANG process. To make it possible for emulated and native code to share data on the same heap we use the exact same tagging scheme in HiPE as in JAM.

However we do not want to use the same calling convention in native code as in JAM, since it would incur an unnecessary overhead to pass all arguments on the stack and maintain the ARGS and VARS registers that HiPE does not need, as it uses registers instead of a stack. Instead HiPE passes the five first arguments in registers and only uses a one word stack frame, containing the previous return address.

We have two stacks for each process, one used in the emulator and one in native code.[5] This solution does have some quirks since there can be catch-frames[6] that are linked with relative offsets on the frame and the whole stack might have to be moved if the process needs more stack space.

Since native code usually runs faster than emulated code we use a higher number of reductions when we in native code check whether the time-slice for the current process is up than we use in the emulator.

This is an imperfect scheme if one mixes execution of emulated code with execution of native code, since the scheduling might be different for mixed code than for only emulated or only native code. Still, this scheme is sufficient for our immediate needs.[7]

[5] A previous ERLANG implementation (JERICO) [12] used a scheme with only one stack. It turned out to be rather complex to maintain the integrity of the stack with two different calling conventions.

[6] A catch-frame indicates where an exception handler is located.

[7] The differences in the sizes of time-slices for native and emulated code does not effect our benchmarks since all executed code is compiled to native code.

3.4.2 Compilation

The HiPE compiler is for the moment a run-time compiler only. That is, it can only compile Erlang code that has been loaded into the JAM emulator, and it can only compile and link into memory; there is no external binary code format. (The intermediate code formats of the compiler can be saved to files as Erlang terms, but this is rather clumsy.)

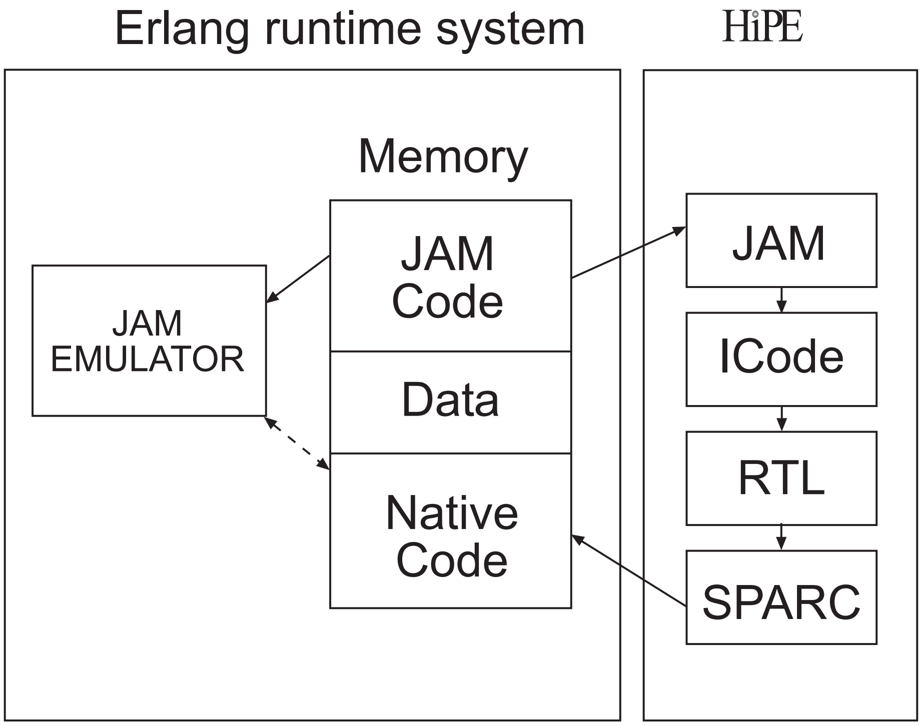


Figure 5: Intermediate representations in HiPE.

The compiler has four intermediate representations; a high level intermediate code called ICode, a general register transfer language in two flavors called RTL(1) and RTL(2), and a machine specific assembly language called SPARC. In Figure 5 the relationship between these representations are shown, note that RTL(1) and RTL(2) are shown as one representation since RTL(2) is a superset to RTL(1).

In the first stage the stack-based JAM code is translated to a register machine with an infinite number of registers. This is done in two steps: first we do a simple translation which introduces some unnecessary anti-dependencies in the code, but these are then removed in the second step, which is a register renaming pass.

In the second stage the ICode is translated to RTL(1). In RTL(1) all calls are assumed to implicitly save all registers. Also reduction counting and garbage collection checks are implicit. After performing some optimizations (such as common subexpression elimination and constant propagation) on RTL(1) the code is translated to RTL(2) where the handling of stack frames, time-slices, and garbage collection checks becomes explicit. A second pass of constant propagation can then be done.

In the third step the code is translated from RTL(2) to SPARC code, where a graph coloring register allocation is performed. The SPARC code is an extended subset[8] of ordinary SPARC assembler represented as ERLANG terms.

In the last step symbolic constants such as atoms and addresses to functions and built-in functions are translated to immediates. Then memory is allocated and the code is linked with the rest of the system.

All destination addresses of calls in native code are hard-coded by the linker. A code server keeps track of all call sites so that they can be back-patched when the implementation of the callee function is changed. This way we can support hot-code loading with no extra cost for normal execution, not only on a per module basis but per function. The extra cost is paid at load time by the native-code server that keeps track of all call sites and performs the back-patching.

[8]Some instructions such as floating point instructions are not implemented as we do not use them. Some extra instructions such as `load_atom` are added to make life easier.

4 Experimental setup

We have conducted our experiments on an UltraSPARC made by Sun Microsystems. The UltraSPARC is a high performance super-scalar processor. It is capable of sustaining the execution of up to four *instructions per cycle* (*IPC*). To achieve this level of *instruction level parallelism* the UltraSPARC uses multiple execution units, a pipelined architecture, branch prediction and prefetching.

This makes it hard for a programmer to predict the performance of the machine code. Many aspects affect the number of instructions that can be dispatched at a time and it is often the case that a program does not execute several instructions per cycle. In many cases the number of instructions per cycle is less than one, and one talks about the number of *cycles per instruction* (*CPI*) instead, which on the UltraSPARC ideally is 0.25.

To help programmers, and especially to help compiler writers, the UltraSPARC has the ability to gather low level performance information. With this information it is possible to find out how well a program is utilizing the hardware.

We have extended our HiPE system so that we can gather this low level information in an easy and efficient manner. We have also extended the HiPE system in other ways to enable other kinds of performance profiling.

In this section we will describe the Sun UltraSPARC architecture, what kind of information the low level performance counters can gather, and the performance extensions made to HiPE.

4.1 Hardware

We run our benchmarks on a 143 MHz single processor Sun Microsystem Ultra 1 Model 140 with 128 MB of memory running Solaris 2.6. We use this system because it is the fastest single processor UltraSparc system at our department. (In the section about performance measurement on UltraSparc (Section 4.2) we describe the problems of using a multiprocessor system when measuring.)

This processor has a hierarchical memory system, several execution units, a nine stage pipeline capable of issuing up to 4 instructions per cycle, and dynamic branch prediction.

4.1.1 Memory architecture

The memory consists of 128 MB of main memory, an *external cache* (level 2 cache) of 512 KB, and two 16 KB on-chip caches. An overview of the architecture

```
On-chip data cache:            On-chip instruction cache:

  sub-block sub-block
      0         1
 +---------+---------+         +--------------------+
 |                   |  ^      |                    | ^
 |         |         |  |      |       SET 0        | | 256
 |                   |  | 512  |                    | v lines
 |         |         |  |      +--------------------+
 |                   |  | lines |                    | ^
 |         |         |  |      |       SET 1        | | 256
 |                   |  v      |                    | v lines
 +---------+---------+         +--------------------+
  <------------------->          <------------------>
     32 bytes line                  32 bytes line
  <-------> <------->              (8  instructions)
  16  bytes 16  bytes
```

can be seen in Figure 6.

The external cache can handle one access per cycle. These accesses are pipelined and after 3 cycles 16 bytes are returned.

The on chip *data cache* is a 16 KB direct mapped cache, organized as 512 lines with two 16 byte sub-blocks per line. A reference to the external cache returns one such sub-block.

The *instruction cache* is a 16-KB pseudo-two-way set-associative cache with 32 byte blocks. That means that there are two sets of 256 lines with 8 instructions in each line. The address of an instruction is conceptually divided into two parts, the *address* part (bits 31 to 5) and the *offset* part (bits 4 to 0). The address part is further divided into a tag (bits 31 - 14) and an *index* (bits 13 to 5).

The UltraSPARC also has a Load Buffer and a Store Buffer. Loads that misses the on-chip data cache are buffered in the Load Buffer until the external cache returns the requested data. In this way the pipeline (see below) will not need to stall if a load misses the cache, unless the result of the load is needed by the other instructions in the pipe. All store instructions are buffered in the Store Buffer (whether they stall or not) this way the pipeline need not stall because of a time consuming stall, unless the Store Buffer is full. In the Store Buffer, consecutive stores may, under certain conditions, be grouped to improve store bandwidth.

4.1.2 Pipelining

Pipelining is a processor implementation technique that exploits parallelism among the instructions in a sequential instruction stream by starting the execution of a new instruction before the execution of the current instruction is finished.

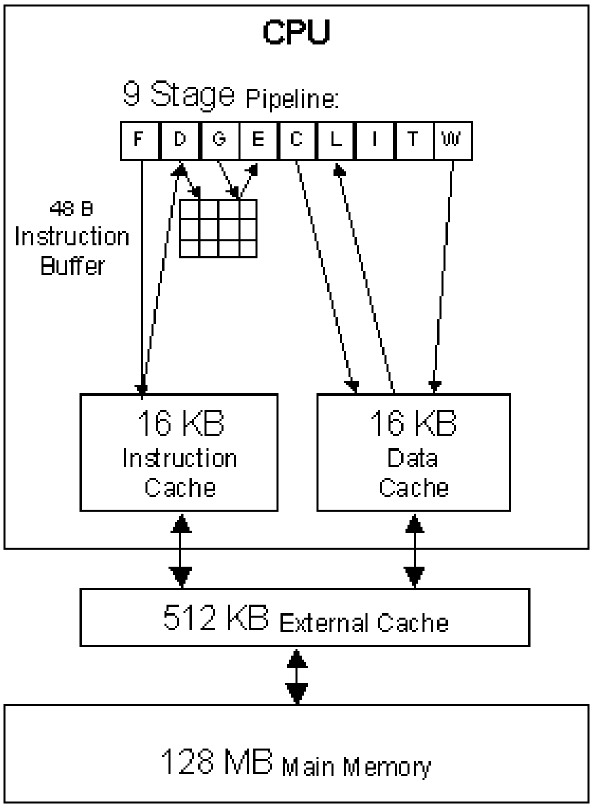


Figure 6: Memory and pipeline architecture on UltraSPARC.

The UltraSPARC 1 has a 4-way SuperScalar design with 9 execution units, 4 integer execution units (*IEU*), 3 floating point execution units (*FPU*), and 2 graphics execution units (*GRU*).

The pipeline is a 9-stage instruction pipeline. The pipeline can be seen in Figure 6, where each letter in the pipeline corresponds to one of the stages as follows:

Fetch (F) – (Pre-)fetches up to four instructions from the instruction cache.

(See Section 4.1.3)

Decode (D) – Decodes the fetched instructions and inserts them into the instruction buffer.

Grouping (G) (or Dispatch) – Groups and dispatches up to four instructions from the instruction buffer (to the appropriate execution units).

Execution (E) – Executes integer instructions and calculates virtual addresses.

Cache Access (C) – Accesses the data cache, and resolves branches.

Load Miss (L) – If a cache miss is detected, the instruction causing the miss is stored in the load buffer.

Integer pipe wait (I) – The integer pipe waits for the floating point/graphics pipe to finish.

Trap Resolution (T) – Any traps are resolved.

Writeback (W) – All results are written to the register files and instructions are *committed*.

A 9 stage pipeline implies that there is a latency of up to 9 cycles for each instruction. Therefore it is important to keep the pipeline full at all times so that at least one instruction finishes in each cycle.

4.1.3 Prefetching

To keep the pipeline full, each instruction fetch fetches four instructions to the instruction buffer. However, if the address for fetching points to one of the three last instructions in an instruction cache line, only one, two, or three instructions are fetched instead of four.

Prefetched instructions are stored in the instruction buffer of at most 12 instructions, until they are sent to the rest of the pipeline.

Instructions can be prefetched from all levels of the memory hierarchy, including the instruction cache, the external cache and the main memory.

4.1.4 Prediction

The UltraSPARC uses both static and dynamic branch prediction. To dynamically predict the outcome of a branch, a two-bit history of the branch is maintained. The bit field is shared between every two instructions in the instruction cache.

The bit patterns in the bit field represents the information *not taken*, *not likely taken*, *likely taken*, and *taken*[17].

This information is used as described in [11]: the prefetch unit interprets the first two states to mean that the branch will not be taken and the last two to mean that the branch will be taken. When a branch is taken it is updated to the *taken* state, unless it is in the *not taken* state, in which case it is updated to the *not likely taken* state. When a branch is not taken it is updated to the *not taken* state, unless it is in the *taken* state, in which case it is updated to the *likely taken* state.

By using static branch prediction the bits can be initialized by the compiler to either *not likely taken* or *likely taken*.

The processor also has *branch following*, the ability to rapidly fetch predicted branch targets. A *next field* associated with groups of four instructions in the instruction cache points to the next instruction cache line to be fetched. The next field points to the next line in the instruction cache for sequential code. If the group contains a branch that is predicted taken then the next field points to the line and offset of the destination of that branch.

4.2 Performance measurement on UltraSPARC

On UltraSPARC processors, low level performance information can be gathered and accessed at run-time. For example, the number of executed instructions and the number of elapsed clock cycles can be counted. Other interesting aspects include the number of cycles spent stalling because of different types of cache misses, and the number of cache references and cache hits.

The UltraSPARC CPU has two registers that are used for performance data. The Performance Control Register (*PCR*) and the Performance Instrumentation Counters (*PIC*). These registers reflect events that happen on a per-processor basis.

The PCR is used to control what to measure; this is done by writing a bitmask to the register. This bitmask tells the processor what two aspects to count. It also tells the processor whether to count events in user mode or in privileged mode, or the sum of events in both modes. The two halves (PIC0 and PIC1) of the PIC register are specialized to measure different aspects.

In our system we can measure:

- Elapsed cycles and issued instructions, giving us the possibility to determine the CPI (cycles per instruction) ratio for our programs.

- Cache references and cache hits for the data cache, the instruction cache, and the external cache.
- The number of cycles the processor spends stalling because of branch misprediction and instruction cache misses.
- The number of cycles spent stalling because the store buffer is full.
- Load stalls, that is, stalls because a loaded value is needed but not present.
- The number of cycles the pipeline is stalled when a load is delayed because an earlier store is incomplete. These stalls are classified as a read after write (RAW) stalls.

4.2.1 Problems

There is no way to clear the PIC register, so we can not control when the register will wrap. Therefore we have to take wraparounds into account when doing our measurements.

Fortunately the effect of this problem is rather easy to spot, and any erroneous values can be discarded.

The performance counters are specific to a particular CPU, therefore we have to run our benchmarks on a single processor machine or bind the process to a specific CPU. Otherwise there is a risk that the operating system would schedule the process to different CPUs at different times thus rendering the measurements totally useless.

The PIC register is a 64-bit register but Solaris 2.6 is not a full 64-bit operating system. This means that only 32-bits of a 64-bits register are saved and restored when a context switch occurs. If we are very unlucky we can get a context switch after we have read the PIC register but before we have saved the value to memory. This could cause that part of the value to become corrupted.

It is improbable that this will happen, especially since we do not run any other 64 bit programs (except for the OS) when we do our measurements. If this problem would occur, then it would not occur at the same place and in the same way at every run. This means that if we make several runs and they all produce similar measurements, we can conclude that we have not been drastically affected by this problem. To handle this in practice, we run the benchmark several times and compute the standard deviation, maximum, minimum and average values for these runs.

The PIC register is process independent. This means that we measure the behavior of all processes running concurrently on the CPU, and not only our benchmark process. This we deal with by running on an unloaded machine. Still we do get some interference from other processes but the effects are small compared to the total measurements. It would be preferable to have an operating system that could keep the performance registers process specific by saving and restoring the values when a process switch occurs.

There is also the risk that some counters are overlapping. The load stall counter for example might sometimes count the same cycle as the misprediction stalls counter. This means that the total number of stalls as compared to the total execution time might be a little bit too high. There is unfortunately no easy way to determine if this has happened.

4.3 Instrumentation of HiPE

In our run-time system we have added some performance instrumentation. Each instrumentation is included or excluded from the system at compile time (of the run-time system).

These instrumentations fall into two broad categories: counters and PIC measures. The counters are just incremented by one each time the execution passes through them. The PIC measures on the other hand are done by first reading the PIC register before an event and than reading it again after the event. The difference between the value before and the value after is then added to an accumulating counter.

All these counters can then be reset or read by calling special BIFs. The PCR can be set by calling a BIF that controls what to measure.

We can also, from an Erlang program, by a call to a BIF read the value in the TICK register that counts cycles.

4.3.1 Counters

Each time a function is called, be that locally, remote, or by a meta-call (apply), a counter for that function is incremented. We can also record each call to a built-in function. For each sent message the program counter of the receiving process can be recorded together with the program counter of the sender.

In the HiPE run-time system there is an interface between the C-code of the emulator/run-time system and native compiled Erlang code. The execution passes through this interface when native code is suspended or needs to perform garbage collection or calls emulated code. The same interface is also used when execution passes from emulated code to native code. Each pass through this interface can be counted.

Each execution of a JAM instruction in the emulator can be counted. For native code we can also turn on basic block profiling. Then we can see how many times each basic block is executed. This way we can see exactly how many times each native code instruction is executed, and hence also which instructions that are not executed at all.

4.3.2 Performance counters

The counter in the PIC register can be accumulated at several points in the HiPE system. The time spent in garbage collection, the time spent in each BIF, the time spent in native code, and the time spent in each time-slice can be measured.

5 Performance on sequential code

Before we start to examine the large programs, let us look at how the compilers behave on a very small sequential benchmark.

The benchmark, *length*, computes the length of a 20,000 element long list 10 times. For each measurement we run this benchmark 20 times (with each system) and compute the average.

The systems we have looked at are HiPE 0.2, JAM 4.5.3 (modified), and BEAM 4.3. We have also done a simple measurement with `erlang:statistic/1` to compare JAM 4.5.3 (modified) with 4.5.3 unmodified and with JAM 4.7.3 to see how our changes have affected the emulator and to see how a more modern emulator behaves. Our modifications to 4.5.3 has made it about 5% slower than the unmodified version and JAM 4.7.3 is about 8% faster than the modified JAM 4.5.3 on this benchmark.

The *length* benchmark consists of two functions `iterate/2` and `len/2`. The function `iterate/2` is responsible for calling the function `len/2` a given number of times (10 in the case of these measurements). The function `len/2` calculates the length of the list in the first argument by using the second argument as an accumulating parameter (see Code 1).

Code 1 (The code for the benchmark *length*)

```
-module(length).
-export([iterate/2]).

iterate(0, _) -> ok;
iterate(X, L) ->
  len(L, 0),
  iterate(X-1, L).

len([_|X], L) ->
  len(X,L+1);
len([], L) ->
  L.
```

5.1 Generated code

We will look at generated code from the tree different compilers in order to get a feeling for how they differ. We will not look at the complete *length* benchmark but at the function `len/2`.

5.1.1 Generated JAM code

The first 13 instructions of the JAM code (see Code 2) is the inner loop of `len/2`. These are the instructions used to traverse the 20,000 elements long list and count the number of elements.

The first instruction (`info(length, len, 2)`) just tells the emulator which function it is executing (`length:len/2`). The second instruction sets the *fail-point* to label 15, this means that if any following test fails execution should continue at label 15. Then room for one local variable (X) is allocated. The first argument is pushed on the top of the stack.

Then a test whether the top of the stack contains a list (actually if it is a cons cell, not the empty list `nil`) is performed. If the test fails execution will continue at the fail point (label 15). This test also pushes the *head* and the *tail* of the list on the top of the stack if it succeeds.

If the test succeeds the head of the list is discarded (poped). Then the tail of the list is saved in variable 0 (X). The failpoint is now removed `commit` so any test that fails will result in an exception. The variable X is then pushed back on top of the stack, followed by the second argument (L) and the integer 1. Then 1 is added to L since they are on top of the stack when the addition operator is executed.

Now the function can recursively be called with X and $(L + 1)$ as arguments. This then goes on until the list is empty and the `unpkList` instruction fails to label 15 where the accumulated parameter L is returned.

Code 2 (The JAM code for the function `len/2`)

```
length_len_2:
  info(length,len,2)
  try_me_else(15)
  alloc(1)
  arg(0)
  unpkList
  pop
  storeVar(0, var,0)
  commit
  pushVar(0,var,0)
  arg(1)
  push(1)
  binop('+')
  enterlocal(length,len,2)

15:
  try_me_else_fail
  arg(0)
  get([])
  commit
  arg(1)
  ret
```

Each JAM-instruction requires at least one load for the instruction, and another load for the address of the code of the instruction followed by a jump. This means that in addition to the actual code that does the work there are at least 36 SPARC instruction in the inner loop of the benchmark. And even if all JAM instructions were as small as the smallest (*pop*, one extra SPARC instruction) there would be 48 SPARC instructions in the loop. But most JAM-instructions are more complex than that. The instruction `enterlocal` executes at least 30 SPARC instructions. In total JAM needs 240 SPARC instructions to execute the inner loop of the benchmark.

5.1.2 Generated BEAM code

The BEAM code also begins with an instruction that indicates which function is being executed but this instruction is not executed in each iteration of the inner loop. The first argument is always in register `x(0)` in BEAM. This register is tested to see if it contains a nonempty list. If not execution continues at label 9 otherwise at the next instruction.

The next instruction reads the head and the tail of the list to registers `x(0)` and `x(2)`. Then the integer 1 is added to register `x(1)` which contains the second argument (L). If anything goes wrong here then an exception is thrown indicating the function beginning at label 39.

Now the registers `x(0)` and `x(1)` contains the arguments x and $(L+1)$ respectively. The instruction `call_only` is then used to check if it is time to suspend the process and jump back to label 3 otherwise.

The loop is then repeated until the end of the list when execution continues at label 9 where the accumulated length L is moved to the return register `x(0)` before the function returns.

Code 3 (The BEAM code for the function `len/2`)

```
len_2:

39:  func_info(length,len,2)
3:   is_nonempty_list(x(0)) failto 9
     x(0), x(2) := get_list(x(0))
     x(1) := arith('+', 1, x(1)) failinfo(39)
     call_only(3)
9:   is_nil(x(0)) failto 10
     x(0) := x(1)
     return
10:  function_clause_error(39)
```

The inner loop is just 4 BEAM instructions, and it only takes 47 SPARC instructions to execute the inner loop for BEAM.

5.1.3 Generated native code

HiPE generates native code from JAM code, making the general structure of the native code similar to the JAM code, but as opposed to JAM, most local values are stored in registers instead of on the stack. The native code does of course become a lot longer than the virtual machine code for JAM or BEAM. We will therefore look at it in several steps, describing each part by itself. First, in Code 4, the inner loop is described, followed by the base case (Code 5). Then we will show how process suspension is handled (Code 6) and finally in Code 7 we show how bad arguments are handled.

Code 4 (The SPARC native code for the inner loop of the function `len/2`)

```
length_len_2:
.length_len_2_13:                 ! External entrypoint
    mov %r8, %r3
    mov %r9, %r5

.length_len_2_1:                  ! Loop entrypoint
    add %r21, 1, %r21                            !  1 Inner
    subcc %r21, 4000, %r0                        !  2 loop
    bge,pn %icc, .length_len_2_2 ! pred: 0.01   !  3
    nop                                          !  4
.length_len_2_3:                  ! No suspension
    srl %r3, 28, %r1                             !  5
    subcc %r1, 10, %r0                           !  6
    bne,pn %icc, .length_len_2_5 ! pred: 0.50   !  7
    nop                                          !  8
.length_len_2_4:                  ! It is a cons
    and %r3, %r27, %r4                           !  9
    sethi 262144, %r1                            ! 10
    srl %r5, 28, %r2                             ! 11
    or %r1, 1, %r1                               ! 12
    lduw [%r4+4], %r4                            ! 13
    or %r2, 1, %r2                               ! 14
    subcc %r2, 1, %r0                            ! 15
    bne,pn %icc, .length_len_2_9 ! pred: 0.01   ! 16
    mov %r1, %r3                                 ! 17
.length_len_2_12:                 ! The arguments are integers
    sll %r5, 4, %r1                              ! 18
    addcc %r1, 16, %r2                           ! 19
    bvs,pn %icc, .length_len_2_9 ! pred: 0.01   ! 20
    nop                                          ! 21
.length_len_2_8:                  ! No Overflow
    srl %r2, 4, %r2                              ! 22
    sethi 262144, %r1                            ! 23
    or %r2, %r1, %r2                             ! 24
.length_len_2_10:                 ! Keep looping
    mov %r4, %r3                                 ! 25
    ba .length_len_2_1                           ! 26
    mov %r2, %r5                                 ! 27
```

This code is in no way optimal, there are optimizations (such as hoisting of loop invariant expressions) that could make each iteration of the loop even smaller. We are doing a quite straightforward compilation of the JAM code to native code, which still gives a considerable speedup.

In native code all tests that are implicit in the code for the virtual machines has to be done explicit. We have to explicitly check whether it is time to suspend the process, whether both arguments to the addition are fixnums, and whether the addition caused overflow. If all this is OK and the argument is a cons cell then we can keep on looping. For HiPE generated native code the inner loop is just 27 SPARC instructions.

At some time the end of the list is reached and then the execution continues at the label `length_len_2_5` (see Code 5). Here we check whether the argument is *nil*, if that is the case the accumulated length is moved to the return register.

Code 5 (Base case of `len/2`)

```
.length_len_2_5:                 ! No cons but is it nil?
    subcc %r1, 9, %r0
    bne,pn %icc, .length_len_2_7 ! pred: 0.01
    nop
.length_len_2_6:                 ! It is nil
    mov %r5, %r8
    jmpl %r15+8, %r0 ! (%r8)     ! Return the result
    nop
```

When traversing a 20,000 elements long list the processes will need to be suspended every now and then, this is done with the code at label `length_len_2_2` (see Code 6).

Code 6 (Process suspension in `len/2`)

```
.length_len_2_2:                ! The time-slice is empty
    st %r3, [%r22+4]            ! Save the process state
    st %r15, [%r22+0]
    st %r5, [%r22+8]
    call swapout_0 ! ()         ! suspend
    add %r22, 12, %r22
    lduw [%r22+-8], %r3         ! Restore process state.
    lduw [%r22+-4], %r5
    lduw [%r22+-12], %r15
    ba .length_len_2_3          ! Keep on working...
    sub %r22, 12, %r22
```

The code must also be able to handle the cases when the function is called with the wrong arguments or the fixnum counter for the length of the list overflows. This code is shown in Code 7 but is never used when the benchmark is run.

Code 7 (Bad arguments handling in `len/2`)

```
.length_len_2_9:                  ! Do general arithmetic
    st %r4, [%r22+4]
    mov %r5, %r8
    mov %r3, %r9
    st %r15, [%r22+0]
    call op_add_2 ! (%r8, %r9)
    add %r22, 8, %r22
    lduw [%r22+-8], %r15
    mov %r8, %r2
    sethi 2359296, %r1
    subcc %r9, %r1, %r0
    lduw [%r22+-4], %r4
    bz,pn %icc, .length_len_2_11 ! pred: 0.01
    sub %r22, 8, %r22
.length_len_2_14:                 ! Did the add succeed?
    ba .length_len_2_10
    nop
.length_len_2_11:                 ! No... (should never happen.)
    ! %r2 = 'badarith'
    mov 0, %r2
    lda bif_exit_1, %r1
    mov %r2, %r8
    jmpl %r1+0, %r0 ! (%r15, %r8)
    nop


.length_len_2_7:                 ! Bad argument (not a list)
    ! %r2 = 'function_clause'
    mov 0, %r2
    lda bif_exit_1, %r1
    mov %r2, %r8
    jmpl %r1+0, %r0 ! (%r15, %r8)
    nop
```

5.2 Speedup

If we for each system S measure the number of clock cycles (T_S) it takes to execute the benchmark we can calculate a rough performance ratio for the system (R_S). We do this with the formula $R_S = T_{JAM}/T_S$ where T_{JAM} is the number of clock cycles for JAM.

The measured average number of clock cycles for each system is: $T_{JAM} \approx 63.32 * 10^6$, $T_{BEAM} \approx 14.07 * 10^6$, $T_{HiPE} \approx 3.93 * 10^6$.

This gives us the performance ratios $T_{JAM} = 1$, $T_{BEAM} \approx 4.5$, $T_{HiPE} \approx 16.1$ as illustrated in Figure 7.

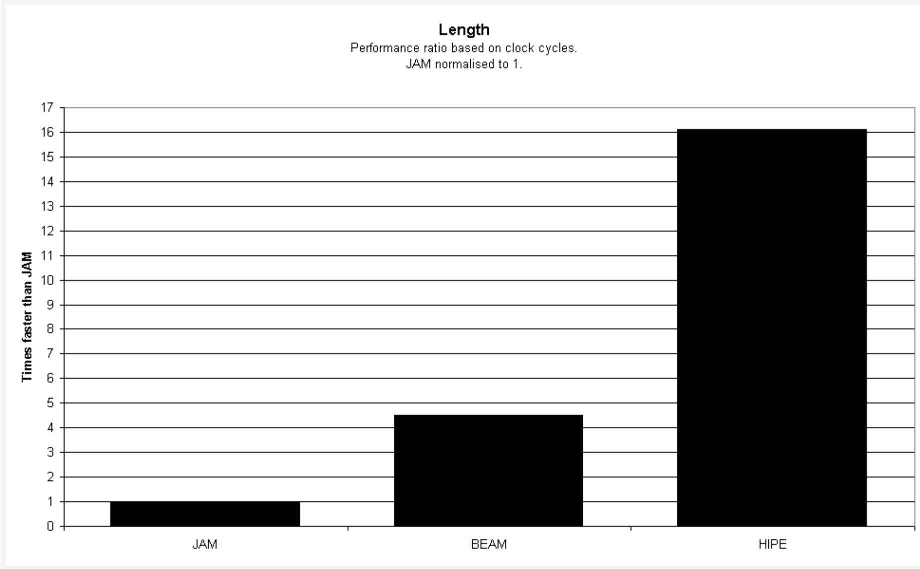


Figure 7: Performance ratio for the benchmark *Length*.

5.3 Instructions and clock cycles

If we look at the number of instructions that are executed and compare that to the number of cycles it takes to execute the benchmark (Figure 8), we can see that only the HiPE compiled program manages to execute more than one instruction per cycle (actually 1.42 instructions per cycle (IPC) or 0.7 cycles per instructions (CPI)).

This is not an optimal result since we are using an UltraSPARC that is capable of issuing up to four instructions per cycle (a CPI of 0.25), but compared with other programs it is a good result. For some C and C++ SPECint95 programs, compiled with *gcc* and executed on an UltraSPARC-II, the CPI varies from 0.83 to 2.13 with a mean of 1.52 for C++ and 1.17 for C [20].

One must bear in mind that a low CPI is not a goal in itself since it can be achieved trivially by executing no-ops. Also, one could argue that a dynamically typed language such as ERLANG does a lot of "unnecessary" work (tagging and untagging) which is easy for the processor to parallelize.

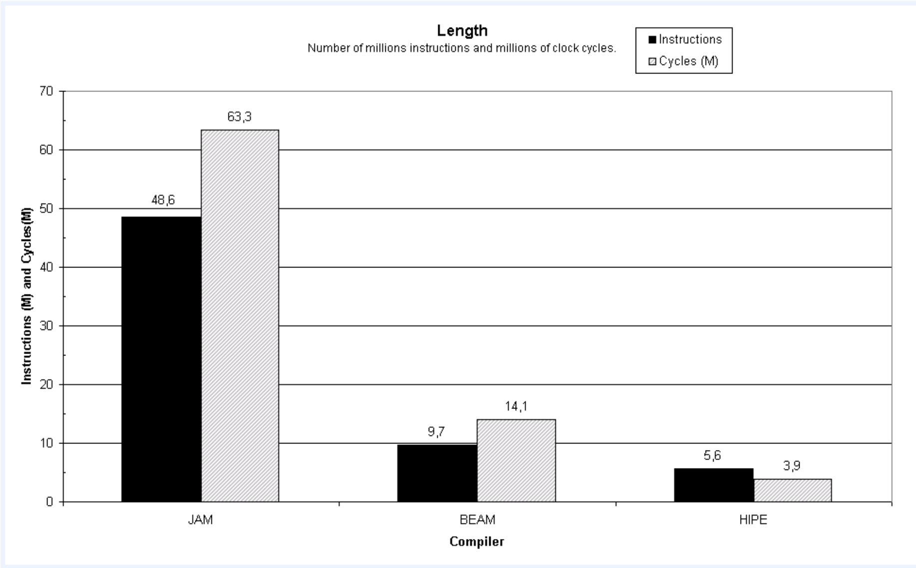


Figure 8: The number of executed instructions (in millions) and execution time (in millions of clock cycles) for the benchmark *Length*.

5.4 Pipeline Stalls

The number of executed SPARC instructions in one iteration of the inner loop of *length* is 27 for HiPE, 47 for BEAM and 240 for JAM. The main reason that JAM needs more instructions than BEAM and HiPE is that JAM that uses a stack has general instructions for function calls and for example for arithmetic. Another reason is that JAM needs instructions to read and write to the stack. BEAM on the other hand does not use a stack and instead has instructions that take several arguments. HiPE can specialize the JAM instructions at compile time and does not use the stack for intermediate values.

Figure 9 shows the total execution time in clock cycles for the three systems. At the bottom of the bars we see the number of million cycles spent stalling because of instruction cache misses. On top of that we see misprediction stalls and load stalls. All other stalls are less than 100,000 cycles and included in the rest of the bar with the non-stalling execution time.

The benchmark is so small that all the native code fits in the instruction cache, hence the time spent stalling because of instruction cache misses is less than 1% of the execution time.

In the JAM emulator the code for one JAM instruction is used in several

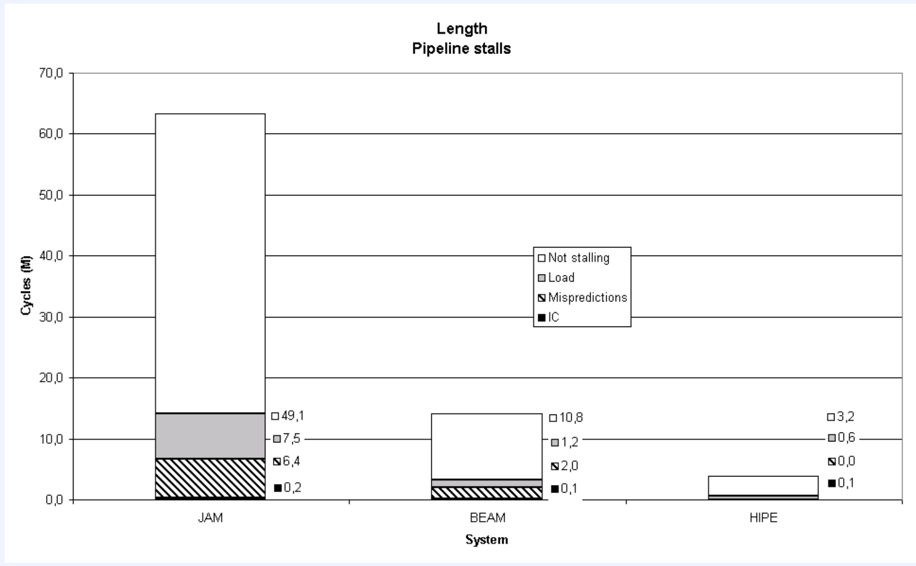


Figure 9: Total execution time and pipeline stalls for the benchmark *Length*.

different contexts rendering the dynamic branch prediction mechanism virtually useless. Therefore the JAM emulator has problems with mispredictions.

If all pipeline stalls were removed from JAM it would still only execute about one instruction per cycle (48.6 M instructions in 49.1 M cycles). HiPE would execute 1.75 instructions per cycles if all stalls where removed. The pipeline can contain from one to four instructions per stage and unfortunately we cannot see from the performance counters how many instructions are in the pipeline while it is stalling. It *might* be that several instructions are stalled for each cycle that JAM is stalling, and for each instruction that HiPE stalls only one instruction is stalled. Therefore we cannot be sure that HiPE not only stalls less but actually has a better scheduled code that can be grouped easier. It does seem very likely though. This does not, however, mean that HiPE has a better scheduler than JAM, it just indicates that in this case the HiPE instructions can be more parallelized.

5.5 Different types of calls

One reason why HiPE shows exceptionally good performance result on `length/2` is that HiPE can compile a tail-recursive call to a real loop if the caller and the callee are the same function. We can show this by making four simple

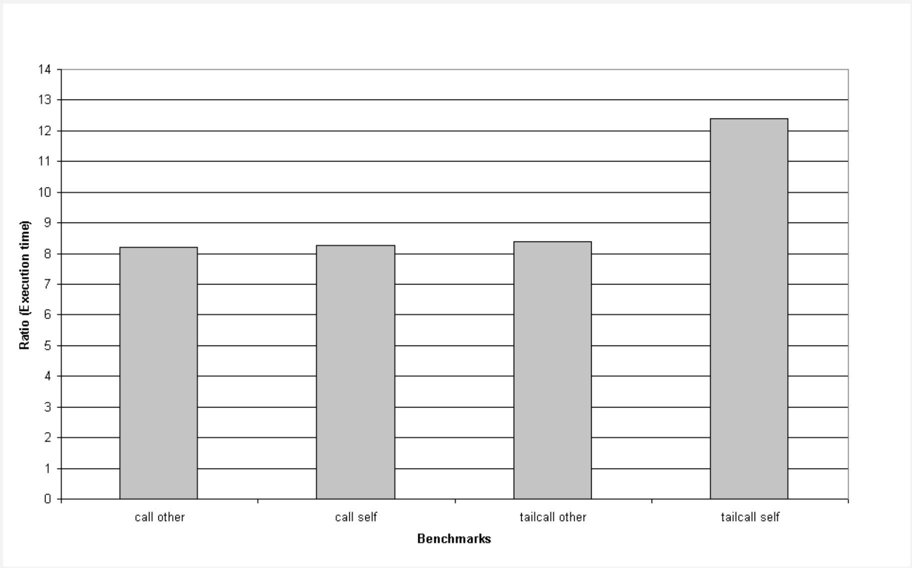


Figure 10: Speedup ratio of execution times for benchmarks with different types of calls in HiPE compared to JAM.

benchmarks that executes a given number of calls.

These four small benchmarks are used to determine the relative speedup for HiPE compared to JAM on different types of calls. There are two "dimensions" to the benchmarks: The first dimension is tail-recursive compared to non-tail-recursive calls, and the second dimension is calls to the same function compared to calls to another function.

The functions only perform some simple arithmetic in order to make the call itself as important as possible. The different benchmarks are not meant to be compared to each other; it is the relative speedup of each benchmark that is to be compared.

```
% ------------------------------------------------
% Here we do a tail call to another function.
% ------------------------------------------------
tail_call_other1(X) when X > 0 ->
  tail_call_other2(X-1);
tail_call_other1(0) ->
  0.

tail_call_other2(X) when X > 0 ->
  tail_call_other1(X-1);
tail_call_other2(0) ->
```

```
  0.

% ------------------------------------------------
% Here we do an ordinary call to another function.
% ------------------------------------------------
non_tail_call_other1(X) when X > 0 ->
  non_tail_call_other2(X-1)+1;
non_tail_call_other1(0) ->
  0.

non_tail_call_other2(X) when X > 0 ->
  non_tail_call_other1(X-1)+1;
non_tail_call_other2(0) ->
  0.

% ------------------------------------------------
% Here we do an ordinary call to the same function.
% ------------------------------------------------
non_tail_call_self(X) when X > 0 ->
  non_tail_call_self(X-1)+1;
non_tail_call_self(0) ->
  0.

% ------------------------------------------------
% Here we do a tail call to the same function.
% ------------------------------------------------
tail_call_self(X) when X > 0 ->
  tail_call_self(X-1);
tail_call_self(0) ->
  0.
```

When we run these benchmarks so that 100,000 calls are made and calculate the speedup ratio between HiPE and JAM as we did for *length* it is even more evident that tail-recursive calls is what HiPE does best (Figure 10). Here we can see that a tail-recursive loop in HiPE can be 12 times faster than in JAM. Other types of calls, those that are not compiled into loops, are only about 8 times faster than JAM.

Things are further complicated since there is a difference between calls within a module and between different modules, but it gives an indication that HiPE will get the best result on the type of functions that are "self tail-recursive"such as *length*. We cannot expect as good results on ERLANG programs with a mix of different types of calls as we could see on *length*.

5.6 Conclusion

The inner loop of *length* can be compiled into a tight (27 SPARC instructions) loop. Since this loop is executed 20,000 times with each branch in the loop going in the same direction each time the branch prediction hardware works very well resulting in a good utilization of the pipeline.

The speedup for HiPE compared to JAM has two reasons: HiPE executes less instructions than JAM, and HiPE utilizes the pipeline better.

HiPE executes less than 12% of the number of instructions JAM executes. If the code would be run on a processor that always executed one instruction per cycle this would result in a speedup of more than 8.6 for HiPE over JAM.

By utilizing the pipeline better, HiPE can execute almost twice (actually 1.87 times) as many instructions per cycle as JAM. We conclude that HiPE utilizes the pipeline better because JAM suffers from misprediction stalls and load stalls, and because more instructions in the native code loop generated by HiPE can be grouped together, and thus executed in parallel by the hardware.

If we take the lower number of instructions to execute together with the better utilization of the pipeline we get a total speedup of 16 (8.6 * 1.87) times for HiPE over JAM.

Unfortunately we can not expect this kind of speedup on all ERLANG programs, since *length* is really just a small loop which is the kind of function that HiPE is getting an extra edge on.

6 The Eddie benchmark

Eddie [5, 21] is the name of an effort to make a flexible and robust web server. This server can be distributed geographically and still maximize the web server throughput. Eddie also supports updates to the server without disturbing the service of the server.

Our benchmark is the HTTP parser in Eddie which parses HTTP *GET requests*. A GET request is what a web browser sends to a server for example when the browser wants a web page. The parser consists of four modules, and a fifth is added for the benchmarking purposes in addition to the ERLANG/OTP standard libraries that are also heavily used.

The benchmark consists of 159 different functions that are called a total of about 30 thousand times. The total JAM code size of the called functions is 13,806 bytes.

The argument to the parser is a list of 30 complex GET requests. Before starting the benchmark these requests are read from a file. The code used to read the requests from file is not benchmarked.

The benchmark starts by spawning a HTTP server which is implemented with the *gen_server* module provided by OTP. In the benchmark there are 5 sends from five different functions.

The benchmark uses 139 different JAM instructions. The total number of executed JAM instructions is 718,364.

6.1 Instructions and clock cycles

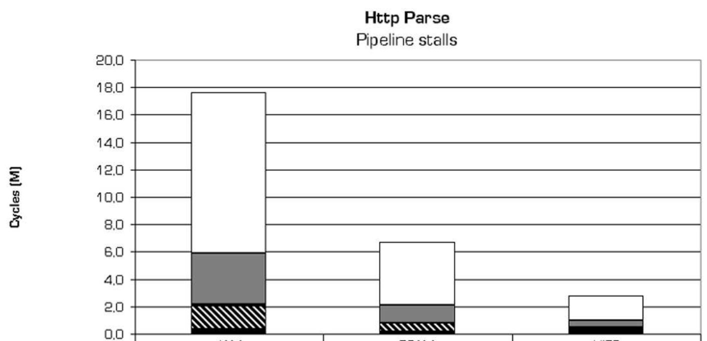
	Cycles (M)	SPARC Instrs. (M)	CPI
JAM	17.64	12.66	1.39
BEAM	6.70	4.35	1.54
HiPE	2.79	2.61	1.07

Table 1: Average execution time in million of cycles, number of millions executed instructions, and CPI for the HTTP parser

If we compute the speedup for BEAM and HiPE as compared to JAM we can see that BEAM is 2.63 times faster than JAM, and that HiPE is 6.32 times faster than JAM. If we compare HiPE and BEAM we can see that HiPE is 2.40 times faster than BEAM.

6.2 Pipeline stalls

Figure 11 shows the total execution time in clock cycles for the three systems as a bar chart. At the bottom of the bars we see the number of million cycles spent stalling because of instruction cache misses. On top of that we see misprediction stalls, read-after-write stalls, load stalls, and store stalls.



	JAM	BEAM	HiPE
Non stalling	11,77	4,57	1,80
Store	0,00	0,01	0,01
Load	3,70	1,30	0,46
RAW	0,11	0,03	0,06
Mispredictions	1,69	0,62	0,10
IC stall	0,37	0,17	0,36

Figure 11: Stalls in comparison to total execution time for the Eddie HTTP parser benchmark.

Both JAM and BEAM spends about 23% of their time stalling because loaded values are needed before the load is completed. For HiPE the load stalls takes about 18% of the execution time.

JAM and BEAM also spend 10% of the execution time stalling because of mispredictions, while the corresponding number for HiPEis 4%.

On the other hand, the percentage of instruction cache stalls is almost 14% for HiPE and only 2% and 3% for JAM and BEAM.

In total these benchmarks have about the same problems with stalls HiPE and JAM stalls about 36% of the time while BEAM stalls about 37% of the time. HiPE makes up for this by having the lowest CPI, executing almost 1 instruction per cycle.

Call type	**Calls**	**%**
Remote tail call	188	0.6%
Remote call	1,218	3.7%
Local call	4,125	12.6%
Local tail call	25,231	77.3%
BIF call	982	3.0%
BIF tail call	545	1.7%
Apply call	6	0.0%
Apply tail call	333	1.0%
Total	**32,628**	

Table 2: Number of different calls in the HTTP parser.

6.3 Different types of calls

There are several different types of calls in ERLANG that all have a different behavior at run time. For JAM, local calls are more effective than remote calls, but for HiPE there is no difference between local and remote calls. HiPE is faster than JAM on all remote calls, since HiPE does not need to look up the destination of the call.

There are also some meta calls (`apply`) and some tail meta calls. The number of each type of call for the HTTP parser benchmark is shown in Table 2.

6.4 Built-in functions

About 4.7% of the calls in the HTTP parser are to built-in functions. JAM spends about 4.4% of the execution time in the 10 built-in functions that are called. The most often called built-in function is `element/2`, which in HiPE is always inlined in the native code. This is a small function and it only stands for about 6% of the total time spent in built-in functions.

The built-in function that takes up most of this time (35 %) is `db_get/2`. This function does a lookup in, and a retrieval from, a database. It stands for 8% of the total number of calls to built in functions.

6.5 Conclusion

HiPEis 6.32 times faster than JAM and 2.40 times faster than BEAM on the HTTP parser. BEAM is 2.63 times faster than JAM. All three systems spends

JAM	Cycles (M)	Instr. (M)	CPI
Not in BIF	16.87	12.15	1.39
In BIF	0.77	0.51	1.51
Total	17.64	12.66	1.39
% In Bif	4.4%	4.0%	

Table 3: Time that JAM spends in built-in functions when running the HTTP parser.

more than 35% of the time stalling, for HiPE it is mainly instruction cache stalls, for JAM and BEAM it is primarily load stalls but also misprediction stalls.

As on *length*, HiPE and BEAM executes a lot less instructions than JAM, and HiPE has a higher IPC than JAM and BEAM. This benchmark uses built-in functions and concurrency. This together with the use of different types of calls probably is the reason that the speedup for HiPE is not as great as it was on *length*.

7 SCCT, the AXD 301 benchmark

We have examined SCCT, a part of an ERLANG program used in AXD 301, a modern ATM switch, developed by Ericsson.

AXD 301 is a new-generation high-performance ATM switching system. The system scales from 10 Gbit/s up to 160 Gbit/s. The program has been designed so that it can be upgraded without stopping the application, and particular care has been taken to ensure high reliability [6].

The program uses about 480 thousand lines of ERLANG code, 330 thousand lines of C code and about 3 thousand lines of Java code. The C code is mostly protocol stacks bought from third party vendors. The ERLANG code is divided into 845 different modules.

In the design of this program the amount of concurrency was kept at a minimum. The designers did not want to have hundreds of thousands of concurrent processes; therefore they decided to use a database in which "virtual" processes are saved so that the ERLANG run-time system does not need to keep all the processes alive.

We have not looked at the whole AXD 301 application but at a relatively small time-critical portion of it, called SCCT. SCCT is responsible for setting up and tearing down connections in the switch. The code we have used is from increment 6 of AXD 301, an earlier version than what is used in the product today.

The benchmark consists of several databases that are setup once initially. The heart of the benchmark consists of several lookups and updates to the databases for each iteration.

The benchmark program is divided into 46 modules. The total size of the JAM code for these modules is 635,270 bytes.

The benchmark uses 11 "background" processes; the startup and initialization of these processes are not measured in our benchmark. The process that runs the benchmark creates another 2 processes for each iteration. We run 100 iterations of the benchmark, so there is a total of 212 processes involved, out of which 14 are alive at a time. That is: 11 (background) + 1 (main) + 2 (new in each iteration). The process communication is also kept at a minimum but for a run of 100 iterations there are 3605 messages sent.

If we look at the run-time behavior of the benchmark we see that only 501 functions in the benchmark are called. And only some parts of the called functions are used. In fact, when we compile the called functions we get a total of 8,880 basic blocks in the code, of which only 2,919 basic blocks are used. This means that if we are not careful we might fill up the instruction cache with a lot of unused instructions.

Out of the 61,234 native compiled instructions 16,463 are actually executed. Since each UltraSPARC instruction is 4 bytes, this means that we have about 64 kilobytes of code that is executed in the benchmark. We will look closer at what this native code does, and where the time is spent.

In the rest of this section we will examine and analyze the results of our benchmarking efforts. We will find that while HiPE was nearly 16 times faster than JAM on the benchmark length, it is only about 1.6 times faster on this benchmark. We conclude that this is because much of the time is spent in built-in functions and because SCCT has fewer tight loops that the HiPE compiler can optimize really well.

On this benchmark the modified JAM emulator is about 10% slower than the unmodified 4.5.3 system. The numbers for JAM that are presented in the rest of the chapter are for our modified JAM system.

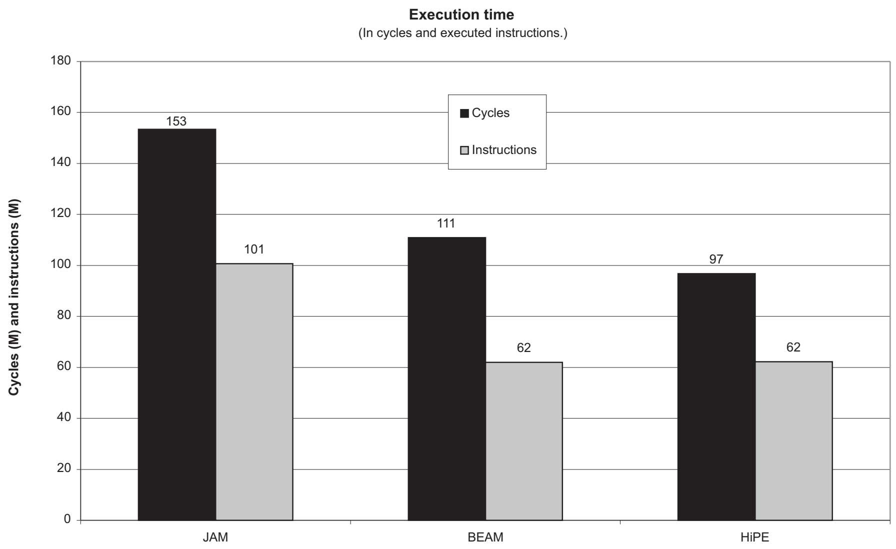


Figure 12: The total running time for SCCT.

7.1 Instructions and clock cycles

Our first measurement counts the number of instructions and clock cycles it takes to execute the benchmark 100 times.

We can see (Figure 12.) that the HiPE system is only about 40 percent faster than the JAM system for SCCT, which is far from the 16 times speedup we saw on *length*.

The HiPE system has lost its ability to run more than one instruction per cycle. The number of cycles per instruction (CPI), where the ideal is 0.25, is 1.79 for BEAM, 1.56 for HiPE, and 1.52 for JAM. This means that, even though HiPE and BEAM executes the same number of instructions, the execution time for HiPE is lower.

We will not take a closer look at the time spent in garbage collection since SCCT only spends about 2 million cycles out of 153 million doing garbage collection.

7.2 Different types of calls

Call type	**Calls**	
remote tail call	13,630	5%
remote call	21,144	8%
local call	60,112	24%
local tail call	36,197	14%
bif call	107,415	43%
bif tail call	10,830	4%
apply call	1,498	1%
apply tail call	1,800	1%
Total	**252,626**	

Table 4: Number of different calls in SCCT.

There are several different types of calls in ERLANG that all have a different behavior at run time. For JAM, local calls are more effective than remote calls, but for HiPE there is no difference between local and remote calls.

There are also some meta calls (using the built-in function `apply`) and some tail meta calls. The number of each type of call for SCCT is shown in Table 4.

Of these only some of the *local tail call*s **may** be tail-recursive calls to the same function. This means that less than 14 percent of all calls can be turned into effective loops by HiPE.

In HiPE we also have a very rudimentary handling of meta calls: we just let the emulator take care of them. This means that HiPE actually is slower than JAM on meta calls, since HiPE has to switch context from native code to emulated code to do a meta call. On the other hand, HiPE is faster than JAM on all remote calls, since HiPE does not need to look up the destination of the call.

7.3 The impact of the memory hierarchy

To see the effects of the memory hierarchy we will look at the pipeline stalls.

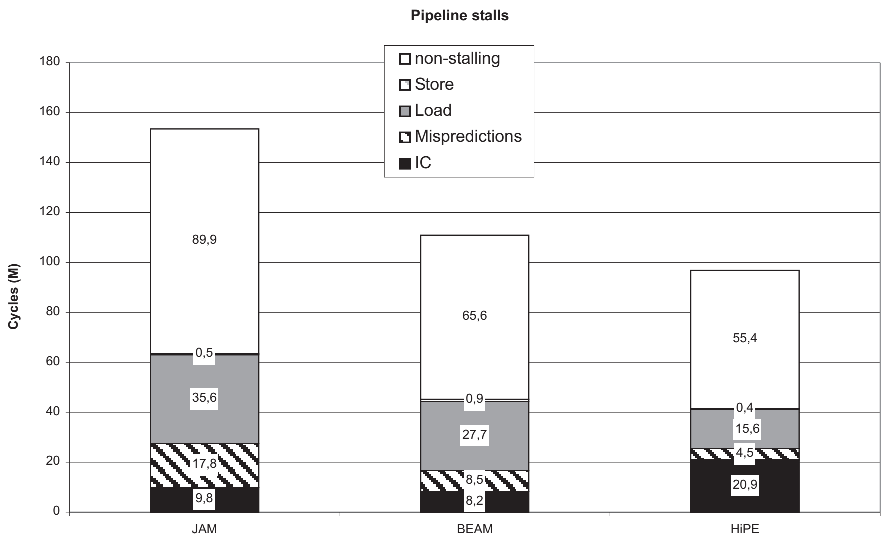


Figure 13: Pipeline stalls in relation to total execution time.

Since the emulated code is compact and the emulator can be small enough to fit in the instruction cache, one could imagine that an emulator could do a relatively better job in avoiding pipeline stalls than native code. But, as we shall see, the gain in instruction cache stalls are counterbalanced by misprediction and load stalls.

The time spent stalling compared to the total execution time, in millions of clock cycles, is shown in Figure 13. As we can see the absolute number of stalls is smaller for HiPE than for JAM.

7.3.1 Misprediction

The number of stalls due to mispredictions is higher for JAM than for HiPE. In JAM the branch prediction hardware is rendered useless since the same emulator code is used in several different contexts.

We can see this on a higher level in the following example:

Example 4 (Dynamic tests that in reality are static.)

```
if
   A > 42 ->
   if
      B > 42 -> never_happens;
     true -> always_happens
   end;
  true -> never_happens
end
```

Even if the test `A > 42` always succeeds and the test `B > 42` never succeeds they will both be implemented by the same piece of code in the emulator making it impossible for the hardware to predict the outcome of the test.

In contrast, when we compile to native code each test will have its own SPARC instruction. If the tests go the same way each time, the dynamic branch prediction hardware will be able to predict the outcome of the tests.

HiPE can also do a good static prediction for some tests such as type tests in arithmetic, which can be assumed to succeed.

One can say that HiPE specializes each JAM instruction with respect to the ERLANG function where it is used. Unfortunately HiPE has to pay for this specialization with increased code volume.

7.3.2 Instruction cache stalls

The number of instruction cache stalls is about twice as many for HiPE as for JAM. But taken together, the number of stalls from mispredictions and stalls from instruction cache misses are about the same for JAM and HiPE.

7.3.3 Load stalls

Load stalls is the main source of pipelines stalls for JAM, since JAM has more data accesses than BEAM and HiPE. Both code and data is stored in memory and JAM uses a stack to store temporary and local variables, whereas HiPE (and to some extent BEAM) uses registers.

On UltraSPARC there is no automatic prefetching, branch prediction, or buffering on data as on native instructions making the choice to have code as data less interesting.

So in absolute numbers the native code is a winner: HiPE stalls about 20 Mc less than JAM. If we on the other hand look at the relative time spent stalling shown in Figure 14 we can see that no matter which implementation we use, about 40% of the time is spent stalling.

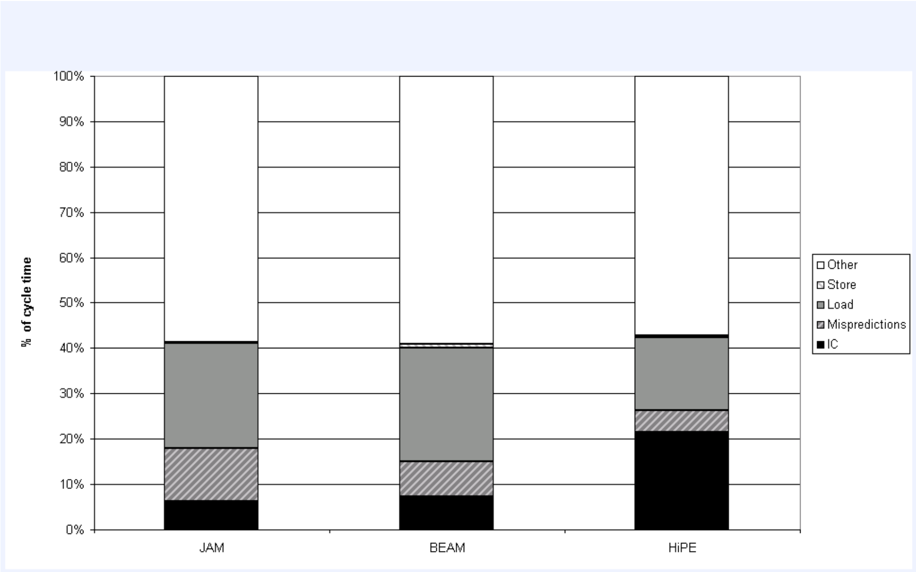


Figure 14: Pipeline stalls in percent of total execution time.

7.4 Concurrency

In the benchmark there is no preemptive suspension; all process switches occur because the running process reaches a `receive` statement with an empty mailbox.

By profiling the message passing we found that 3605 messages were sent from 18 different places in the code. What is more interesting is that for each of these 18 points the receiver of the message was always waiting for that specific message. That is for each of the 18 types of messages the receiver was always at the same point in the code with an empty mailbox. This kind of behavior is more thoroughly described in [14].

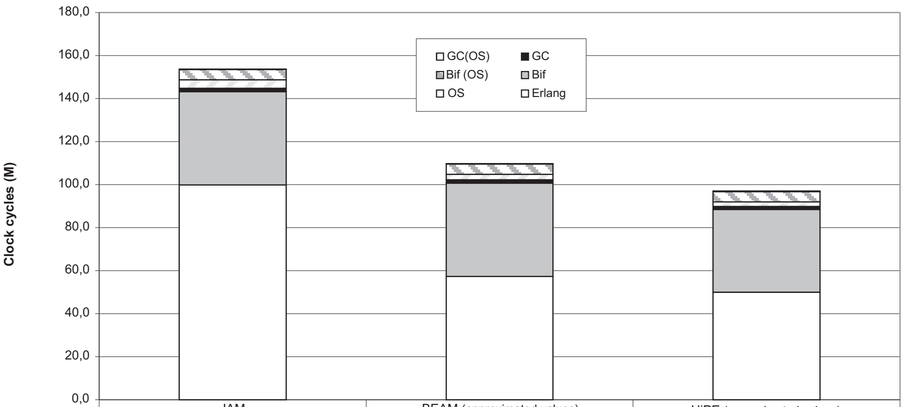
7.5 Built-in functions

The HiPE system uses the same built-in functions (BIFs) as the JAM system.[9] We have made measurements in the JAM system since all calls to built-in functions from JAM goes through the "call-bif" instruction making it easy to measure these calls. In JAM about 49 million cycles are spent in built-in functions,

[9] When we talk about BIFs in this paper we refer to the built-in functions that are implemented as C functions in JAM system 4.5.3.

that is about 32 percent of the total execution time.

In JAM there are 118,245 calls to 26 different built-in functions in the benchmark. Since the benchmark is executed 100 times there are several BIFs that has an even hundred of calls to them. Many built-in functions are only called a couple of times in each iteration and are uninteresting.



	JAM	BEAM (approximated values)	HiPE (approximated values)
GC(OS)	0,2	0,2	0,2
Bif (OS)	4,8	4,8	4,8
OS	4,0	2,5	2,1
GC	1,5	1,5	1,5
Bif	43,4	43,4	38,3
Erlang	99,8	57,3	50,1

Figure 15: The number of million cycles spent in different types of code when executing SCCT. The values for Built-in functions for HiPE and BEAM are approximated, as well as the GC-times.

Since all three systems uses the same garbage collector and the same built-in functions we can approximate the time spent in built-in functions and garbage collection for HiPE and BEAM to the same as for JAM. Since the built-in `element/2` is inlined directly in HiPE we remove the time JAM spent in `element/2` from the approximated time spent in built-in functions (see Figure 15).

If we remove the time spent in the operating system, in built-in functions, and in the garbage collector from the total execution time we get the time spent executing ERLANG code. This is 100 Mc for JAM, 57 Mc for BEAM and 50 Mc for HiPE. The time HiPE spend in ERLANGcode are the time we can affect with optimizations in the compiler. For HiPEthis time (50 Mc) is only about 52% of the total execution time (96.8 Mc).

By calculating the speedups on the time spent in ERLANG code as opposed to on the total execution time we get that HiPE's speedup over JAM is 2.0 and over BEAM it is 1.1. The speedup for BEAM over JAM is 1.9.

It is interesting to compare the total number of calls to BIFs (118,245) with the number of calls to functions (134,321). The built-in functions stand for 32 percent of the execution time and 47 percent of the calls. It is therefore unfortunate that in HiPE calls to built-in functions cost more than calls to functions. (Calls to built-in functions in JAM are even more expensive.)

The most called BIF, `element/2`, is implemented in native code and inlined by HiPE so in that case the overhead for the call is removed.

BIF	**Calls**	**Inst. (M)**	**Mc**	**Stalls (Mc)**				
				Load	**RAW**	**Store**	**Misp**	**IC**
binary_to_list/1	2,502	0.6	1.0	0.4	0.2	0.0	0.0	0.0
db_get/2	6,096	10.2	13.7	2.7	0.0	0.0	0.9	0.6
db_get_element/3	2,400	1.0	2.0	0.5	0.0	0.0	0.1	0.2
db_match/2	300	0.4	0.8	0.1	0.0	0.0	0.0	0.2
db_put/2	1,318	11.4	14.0	2.2	0.0	0.1	1.0	0.7
db_update_counter/3	1,012	0.4	0.9	0.2	0.0	0.0	0.0	0.2
element/2	80,363	5.1	5.1	0.5	0.0	0.0	0.3	0.2
list_to_binary/1	2,402	3.1	5.0	0.8	0.0	0.0	0.3	0.7
setelement/3	9,316	2.2	2.5	1.0	0.4	0.1	0.1	0.1
spawn_link/3	200	0.6	1.4	0.3	0.0	0.0	0.1	0.2
split_binary/2	1,902	0.4	0.7	0.2	0.0	0.0	0.0	0.1
Sum:	**107,811**	**35.4**	**47.0**	**8.9**	**0.6**	**0.2**	**2.9**	**3.1**
% of BIF total:	91%	98%	96%	96%	96%	99%	96%	88%

Table 5: Pipeline stalls for BIFs running more than 0.5 Mc in SCCT on JAM. The percentages at the bottom shows how many percent these 11 BIFs stand for as compared to all BIFs executed by SCCT.

7.5.1 Pipeline stalls for built-in functions

In Table 5 we can see the pipeline stalls for the built-in functions in JAM that take more than 0.5 million cycles to execute. The columns show the name of the BIF, the number of calls to the BIF, and the number of millions of instructions executed while in the BIF, the number of millions of cycles executed while in the BIF, the number of millions of cycles spent stalling (because of load stalls, read-after-write stalls, store buffer stalls, misprediction stalls, and instruction cache stalls, respectively). As can be seen in the bottom line of the fourth column, these eleven BIFs stand for 96 percent of the execution time spent in built-in functions.

We can see that all these built-in functions have problems with load stalls; they are stalling 9–45% of their execution time.

The built-in functions `binary_to_list/1` and `setelement/3` have the highest percentage of load stalls; they also stand for almost all read-after-write stalls (RAW). In Section 8.2.3 we will explain where these read-after-write stalls come from and how to eliminate them.

7.5.2 Built-in database functions

More interesting in terms of absolute performance are the two built-in functions `db_get/2` and `db_put/2`. They stand for 57 percent of the time spent in built-in functions, and 18 percent of the total execution time. These BIFs are used to manipulate a RAM database that resides outside the process heap. The database is implemented as a hash table with buckets implemented as linked lists. These lists sometimes need to be searched through and data needs to be copied to and from the process heap. This can take some time and cause some stalls, but not more than about 30 percent of the execution time of the database functions is spent stalling; this is less than the JAM system as a whole.

7.5.3 Conclusion

The SCCT benchmark is a large benchmark, the size of the executed native code is 64KB, and the total size of the 501 called function is in native code 239 KB.

The execution times are 153.4, 110.9, and 96,8 million clock cycles for JAM, BEAM, and HiPE respectively. Thus HiPE is only about 40% faster than JAM on SCCT, but if we take into account and discard the time spent in built-in functions, in the garbage collector, and in the operating system then we can say that HiPE is 2 times faster than JAM.

BEAM executes just as many native code instructions as HiPE. There are several reasons for this:

- The instruction set for BEAM is more powerful and better designed than JAM's instruction set.
- BEAM has a more powerful compiler than JAM.
- BEAM has a better handling of pattern matching than JAM.

- BEAM uses registers instead of a stack and do not have to shuffle data back and forth to the top of the stack.
- BEAM has a more effective way of saving local variables during function calls than HiPE.

HiPE has problems with instruction cache stalls (20.9 Mc) while JAM and BEAM are troubled by load stalls (35.6 Mc and 27.7 Mc) and misprediction stalls (17.8 Mc and 8.5 Mc). All three systems spends about 40% of their total execution time stalling.

This indicates that even though HiPE runs into problems with the instruction cache because of the size of the program, HiPE do not suffer more from this than JAM and BEAM suffers from other types of stalls. The main reason that HiPE do not get the same speedup as on *length* is that much of the time is spent in code outside its control, such as built-in functions and garbage collection.

8 Future work

In this section we discuss what the HiPE group will do in the future. First we look at the areas that needs further investigations (Section 8.1). Then we describe the areas we feel are important to address in order to get better execution times in HiPE (Section 8.2). Finally we look at some optimizations that we would like to examine in order to see if they would be beneficial to HiPE (Section 8.3).

8.1 Further investigation

There are (some) questions that are raised but not answered by this report. Here we will look at a few of them and speculate in what the answers might be.

8.1.1 Why is BEAM so much faster than JAM?

In this section we will try to come up with some explanations to why BEAM is so much faster than JAM. It will be mostly speculations though, since it has not been a goal within this investigation to answer this question. There are probably several reasons why BEAM is faster than JAM. BEAM has a more advanced compiler and a more advanced virtual machine. The measurements show that:

- The percentage of instruction cache stalls is less for BEAM than for JAM on SCCT (and about equal on the http parser).
- The percentage of mispredictions for BEAM is less than for JAM on SCCT (and about equal on the http parser).
- The percentage of load stalls is less for JAM than for BEAM on SCCT (on the http parser BEAM has a slightly smaller percentage of load stalls than JAM).
- BEAM has a tiny bit of store buffer stalls. (JAM none)
- BEAM is considerably faster than JAM overall.

The BEAM instructions are more powerful than the JAM instructions so the number of instructions needed for a particular task is less for BEAM than for JAM. This results in less native code to execute for BEAM than for JAM, since there is an overhead for each abstract machine instruction to execute. JAM also has to execute a lot of stack manipulating instructions whereas BEAM can operate directly on the registers where the arguments are stored. This results in an even smaller number of abstract machine instructions for BEAM compared to JAM. This can at least partly explain both the difference in instruction cache misses and the overall speedup.

BEAM has less mispredictions, possibly, because it has many different instructions, while JAM has just a few general instructions. The risk of getting the same instruction twice in a tight loop is lower for BEAM than for JAM. This has two effects.

The first effect has to do with the threaded implementation of the emulators. Both JAM and BEAM are threaded so the implementation of each virtual machine instruction ends with a jump to the next instruction. If a BEAM instruction only occurs once in a loop, it is possible for the hardware to predict the destination of the jumps between instructions in the emulator. JAM probably has several `pop` and `push` instructions in each loop, if each `pop` is followed by a different JAM instruction, then the hardware will prefetch the wrong SPARC instructions.

The second effect is similar. The risk of having the same test instruction for two tests going in different directions is also lower for BEAM than for JAM, since BEAM has different instructions for example depending on which register to test.

Pattern matching is compiled more efficiently in BEAM than in JAM. Let us look at an example. In the simple pattern matching example (Example 5) we have four different patterns that are mutually exclusive, which means that we could test them in any order we want. The best would of course be to group all clauses containing a tuple together (in rough pseudo code):

- 1. Is *arg0* a tuple then 2a else 2b.
- 2a. case element(1, *arg0*) of
  - `a`: return(element(2, *arg0*))
  - `b`: return(element(2, *arg0*))
  - `d`: return(element(2, *arg0*))
  - default: fail.
- 2b. is *arg0* the atom `c` then return(`c`) else fail.

Example 5 (Erlang code for matching a simple pattern)

```
test({a, V}) -> V;
test({b, V}) -> V;
test(c) -> c;
test({d, V}) -> V.
```

Unfortunately neither JAM nor BEAM does this. JAM just tests each clause by itself until a match is found (Example 6). BEAM groups similar patterns together as long as no "non-similar" patterns come in between (Example 7). The method of BEAM is of course more effective than that of JAM so that should explain some of the overall speedup for BEAM. (It also partly explains why BEAM can get quite close to the performance of HiPE. Since ERLANG functions tend to rely heavily on pattern matching it is important to have a good pattern matcher, but HiPE is stuck with the naive pattern matcher of JAM.)

Example 6 (JAM code for Example 5)

```
   info(pattern,test,1)     ; This is the function pattern:test/1
   try_me_else(22)          ; Set up label 22 as the fail point
   alloc(1)                 ; Make room for a local variable (V)
   arg(0)                   ; Get the first argument
   unpkTuple(2)             ; Is it a tuple of arity 2? (no -> 22)
   get(a)                   ; Is the first element the atom a?
   storeVar({0,{var,0}})    ; Bind variabel V to element 2
   commit                   ; Remove the fail point 22
   pushVar({0,{var,0}})     ; Get vaiable V
   ret                      ; Return (V)
22:                         ; Label 22 (the first fail point)
   try_me_else(23)          ; Set up label 23 as the fail point
   alloc(1)                 ; Make room for a local variable (V)
   arg(0)                   ; Get the first argument
   unpkTuple(2)             ; Is it a tuple of arity 2? (no -> 23)
   get(b)                   ; Is the first element the atom b?
   storeVar({0,{var,0}})    ; Bind variabel V to element 2
   commit                   ; Remove the fail point 23
   pushVar({0,{var,0}})     ; Get vaiable V
   ret                      ; Return (V)
23:                         ; Label 23 (the second fail point)
   try_me_else(24)          ; Set up label 24 as the fail point
   arg(0)                   ; Get the first argument
   get(c)                   ; Is it the atom c? (no -> 24)
   commit                   ; Remove the fail point 24
   push(c)                  ; Make the atom c
   ret                      ; Return (c)
24:                         ; Label 24 (the third fail point)
   try_me_else_fail         ; No fail point (execption in stead)
   alloc(1)                 ; Make room for a local variable (V)
   arg(0)                   ; Get the first argument
   unpkTuple(2)             ; Is it a tuple of arity 2? (no -> fail)
   get(d)                   ; Is the first element the atom d?
   storeVar({0,{var,0}})    ; Bind variabel V to element 2
   commit                   ; Remove the fail point 24
```

```
pushVar({0,{var,0}})     ; Get vaiable V
ret                      ; Return (V)
```

The pattern matching in JAM is very much inspired by the one used in the WAM. For example, JAM uses *fail points* to keep track of where to continue execution when a test fails. This means that the test instruction does not have to contain information about where to continue execution if the test fails.

As can be seen in Example 6, JAM has to test if the argument is a tuple, for each clause, then it has to read the first element of the tuple from the heap, for each clause, and test it against an atom. The only exception is of course the third clause where it can test against the atom `c` directly.

Example 7 (BEAM code for Example 5)

```
44:
    func_info('pattern','test',1)
1:
    ifnot is_tuple(x(0)) then 6
    ifnot arity(x(0)) == 2 then 6
    x(1) := get_tuple_element(x(0), 0)
    ifnot x(1) == 'a' then 2
    x(2) := get_tuple_element(x(0), 1)
    x(0) := x(2)
    return
2:
    ifnot x(1) == 'b' then 3
    x(2) := get_tuple_element(x(0), 1)
    x(0) := x(2)
    return
3:
6:
    ifnot x(0) == 'c' then 4
    x(0) := 'c'
    return
4:
    ifnot is_tuple(x(0)) then 5
    ifnot arity(x(0)) == 2 then 5
    x(1) = get_tuple_element(x(0), 0)
    ifnot x(1) == 'd' then 5
    x(2) := get_tuple_element(x(0), 1)
    x(0) := x(2)
    return
5:
    function_clause_error(44)
```

In BEAM each test instruction contains information about where to continue execution. The instruction `ifnot ... else` is not the actual BEAM instruction but used here for clarity of the code. This means that each type of test is coded as a different instruction, and not as it might seem here as just different arguments to the `ifnot ... else` instruction.

As can be seen in Example 7, BEAM can group the two first clauses together since they both are tuples, but then it gets confused by the fact that `c` is an atom and not a tuple. In the fourth clause, BEAM must, again, check if the argument is a tuple and load the first element if it is.

8.1.2 Emulation versus native compilation

One aspect that we feel needs further investigation is the advantages and disadvantages, to execution speed, of native code as compared to emulated code.

There are also other aspects of emulated code, such as easier tracing and debugging, smaller external formats, and ease of porting to other platforms, but we are interested in the execution times.

The argument that a small emulator can fit entirely in the instruction cache and that it therefore would have an advantage against clumsy native code seems hollow. It is true that the emulator does not have trouble with instruction cache misses, but it does have trouble with pipelines stalls because of mispredictions and load stalls.

There might be emulator implementation techniques that can remedy this, and this would be interesting to investigate.

8.1.3 Are there unnecessary calls to built-in functions?

The built-in functions are responsible for a big percentage of the execution time, and some of them might be hard to optimize further. It would therefore be interesting to know if some of the calls to these functions could be eliminated.

The short answer is yes. We know that some tuple operations could be grouped. For example some `record` operations in ERLANG are compiled to several consecutive `setelement/3` calls. These could all be done at once saving much overhead. Even though there are many calls to `setelement/3` and `element/2` that could be grouped or removed, they do not stand for a considerable amount of the time spent in built-in functions.

We would like to know how it is with the more time consuming database functions, maybe some of those are redundant or could be grouped if there are several updates to the same database record.

8.1.4 Major timeslice effect on JAM cache

There is one disturbing effect that we have discovered but not been able to explain. This is the effect on the *length* benchmark of increasing the size of time-slices in JAM.

An increased time-slice resulted in an increase in both instruction cache stalls and in load stalls, this could be expected. When we increased the time-slice so that the process did not get suspended at all during the *length* benchmark, this resulted in a reduction in the instruction cache stalls but also in an increase in load stalls, while the total execution time was unchanged. We have not been able to explain this behavior but neither have felt it important enough to investigate further yet. It would be interesting to investigate this further so that complete understanding of the problem could be reached.

8.2 How to improve HiPE

Here we will present some optimization techniques that we think are important in order to get increased performance in HiPE. One thing that we have seen is that even though the few low level optimizations that HiPE performs are effective on a small tight loop, the impact on larger programs is a lot smaller. We have also seen that a lot of time is spent in built-in functions. There are two ways to decrease the time spent in built-in functions. The first is to make the built-ins themselves more effective. The second is to reduce the number of calls to built-ins by making the calling code more effective and remove redundant calls.

Most of the optimizations we describe here have already proved their usefulness for other languages and we feel confident that they will also work well for ERLANG.

8.2.1 The front end

Today HiPE is using the same front end as JAM. This means that we have inherited the naive pattern matching, and that there basically are no high level optimizations in HiPE. One way to rectify this would be to use the BEAM front end, but even though it is better than JAM it still does not contain all high-level optimizations we would like to have in HiPE.

As we have seen HiPE is quite effective on the very small *length* benchmark but not as good on the larger ones. Today HiPE works on extended basic blocks or at most whole functions when optimizing. We belive that a better job could be done if HiPE had a large scope to optimize. This is hard and expensive (measured in compile time) on the low level, but easier on a higher level, since the size of the code is smaller on a higher level than on a lower.

A lot of time is spent in some very expensive built-in functions. It might be hard to optimize these functions but it might be that some of the calls to these function are redundant. This might be recognized by a higher level optimization and redeemed.

A lot of work in the native code is spent on type checks. This is not directly evident from the measurements presented in this report. Though it is not hard to imagine that a dynamically typed language would show such a behavior and when we have studied the produced native code we have seen this. It would be desirable to have an analysis that could guarantee that certain values have a certain type, in order to remove the type tests.

It would also be interesting to have a new common high level intermediate language that would allow us to express high level optimization in an easy way, and to share these with other ERLANG implementations.

Therefore we think that we need a new front end.

Core Erlang A new intermediate language, *Core Erlang*, has be developed in order to get a simpler and cleaner language to perform high level optimizations on. This language has been developed by the HiPE group in cooperation with OTP and the Computer Science Lab at Ericsson. One of the goals of Core ERLANG has been to provide a common platform for compiler optimizations of ERLANG in order to facilitate a faster technology transfer between industry and academia. Much of the work of the design and implementation of Core Erlang has been done by Richard Carlsson.

Core Erlang has the same expressive power as ERLANG but is much simpler from the view of a compiler writer. It is fairly easy to translate an ERLANG program to a Core ERLANG program and vice versa.

Today the HiPE group has an ERLANG to Core ERLANG translator, a Core ERLANG to ERLANG translator, and soon also a Core ERLANG to ICode translator.

Module Merging As we mentioned earlier we would like to get larger scopes for the compiler to optimize. One way to do this without having to write a complete inter-functional optimizer is by inlining. Unfortunately the hot code loading semantics of ERLANG makes this hard, except for within a module.

One often used way to implement an abstract datatype in ERLANGis to put all primitives for the datatype in its own module. This can make a simple access to an element in such a data structure more expensive than necessary, since a remote call has to be done. We would therefore like to have inlining across module boundaries.

In order to facilitate this we think that a technique that we call *Module*

Merging could be useful [16]. Module Merging is the ability to take two (or more) ERLANG modules and merge the code in those modules in such a way that remote calls between the modules can be replaced with local calls. Module Merging will also preserve the code replacement semantics of ERLANG so that if a new version of the code for any of the merged modules is loaded, then the new code will be used instead of the old merged code.

After Module Merging is applied to two modules it will be possible to also inline function calls that before were between modules.

Inlining Function calls are expensive, at least if they are not tail recursive calls where the caller and the callee are the same function. HiPE would get rid of some call sites by inlining, and in mutually recursive functions, one of the functions could be inlined and the other turned into a self-recursive function.

Inlining will also pave the way for further optimizations that now are local to a function.

Pattern Matching Compilation Today HiPE uses the same pattern matching as JAM, since we use the same front end. As we have seen, this can result in a lot of unnecessary work. Since pattern matching plays such a central role in ERLANG we think it deserves a better treatment.

An algorithm for effective pattern matching compilation is described by Philip Wadler in [19]. Richard Carlsson has implemented a pattern matching transformation on Core ERLANG based on this algorithm. It has already shown very promising results on some informal measurements.

Patterns in `receive` statements can not be optimized on the Core ERLANG level. This is because the pattern matcher needs access to messages in the mailbox without removing them, a functionality that neither ERLANG nor Core ERLANG provides.

The HiPE intermediate code, ICode, provides the means to look at a message in the mailbox and only extract the message if it matches any pattern, therefore the Core ERLANG to ICode compiler will be able to optimize patterns in `receive` statements.

Type Analyzer As noted before, a lot of work is done to check that arguments to primitive operators have the right type. This is a price that ERLANG have to pay for the powers of being dynamically typed. We believe that this price sometimes could be avoided.

By performing abstract interpretation on Core ERLANG it should be possible to obtain information about types in ERLANG programs. This information can then be used by the compiler, for example to remove redundant type checks.

One "side effect" of the type checking is that some bugs in the analyzed program can be detected. This is a result that we already have seen with the type analyzer being developed by Sven-Olof Nyström.

Partial Evaluation One often mentioned benefit of ERLANG is that it comes with a large standard library and an even larger library provided by OTP. These libraries provides general solutions to common problems, which makes development in ERLANG very rapid.

Generality does of course come with a cost in performance. Fortunately this does not have to be the case. With partial evaluation [15] these general library functions can automatically be specialized with regards to the application at hand. We believe that partial evaluation together with Module Merging will do wonders to the generic functionality provided by OTP.

Richard Carlsson is working on a partial evaluator for Core ERLANG.

8.2.2 The run-time system

Even though we feel that the front end is where we can gain the most in performance there are some adjustments that could be done to make the run-time system better.

The tagging schemes in JAM and BEAM are not ideal, especially not for the UltraSPARC architecture. In HiPE today the tags are in the four most significant bits. We think that a scheme with sub tags in the least significant bits would make pointer dereferencing and arithmetic cheaper. Mikael Pettersson has designed and implemented such a scheme for ERLANG.

Since OTP constantly is improving the ERLANG system and today there is a free open source version of ERLANG, it would be nice if HiPE could take advantage of the improvements and at the same time make HiPE available to others. Therefore it would be interesting to integrate HiPE with open source ERLANG.

One other aspect of the run time system is the relation between the processes heaps and the garbage collection algorithm. We will talk more about this below.

8.2.3 Built-in functions

Since the built-in functions are so important (32% of the execution time for SCCT in JAM) they deserve a closer look. The small ones could be inlined directly in the native code, others could be rewritten in ERLANG, ICode, or RTL.

The built-in database Today the database is placed outside the heap and data has to be copied to and from the database. If the data in the databases could be left on the heap a lot of time could be saved.

RAW stalls in `setelement` The built-in function `setelement/3` takes a tuple (T), a position (N), and a new element (E), and returns a new tuple identical to T except that the Nth element is replaced by E. This is done by first copying the entire original tuple, with the following C code:

```
res = make_tuple(BIF_P->htop);   /* Get hold of new tuple */
*BIF_P->htop++ = *tuple_ptr++;   /* Copy arity */
while (size--)
  *BIF_P->htop++ = *tuple_ptr++; /* Copy each element */
```

The address to the heap top is stored in the process structure in each iteration and that value is then read in the next iteration. This causes the load to stall since it will be scheduled together with (or directly after) the store, and that store will take at least one clock cycle to complete. This type of stall is called a read-after-write stall or RAW-stall.

If we rewrite `setelement/3` with a local variable instead of `BIF_P->htop` then the read-after-write stalls are eliminated and the total execution time for `setelement/3` in the SCCT benchmark drops from 2.5 Mc to 1.7 Mc, a gain of about 30%.

The absolute performance gain for SCCT would be small since the total time in `setelement/3` is only 2.5 Mc out of 153 Mc.

This implementation glitch is fixed in later versions of ERLANG where all the built-in functions have gotten a thorough overhaul. This shows that these low level measurements can be of real use when tuning time critical code.

As we could see with the built-in function `setelement/3` even the really small built-in functions need to be closely examined to make sure they are effectively coded.

8.2.4 Standard optimizations on intermediate code

As we have seen the low level optimizations that HiPE performs today do not have the same impact on large programs as on small ones.

Since there are almost no loops in the intermediate code of HiPE today, there are many standard optimization techniques that would not be very effective. Inlining should introduce some more loops making it interesting to add loop optimization to HiPE.

Inlining would also make other optimizations that work within a function more effective. With inlining it would be interesting with a more advanced constant propagation, constant folding, etc, than what HiPE has today.

There are also a lot of other standard optimizations that could be interesting, such as inter-procedural register allocation.

8.2.5 The back end

Since HiPE, as all three systems, spends about 40% stalling (at least for SCCT) it is important with low level optimizations that can decrease the number of cycles per instruction.

There are three techniques that we think can do this, instruction scheduling, prefetching of code and data, and possibly the use of predicated execution.

Since so much time is spent in built-in functions and the run-time system the absolute gain will not be that big even if all stalls in native ERLANG code could be eliminated. Therefore we would not like to put to much effort into this at the moment.

It would also be nice to have more back ends than SPARC, at least a x86/Linux back end and maybe a x86/NT back end.

One way to achieve both the goal of a better back end and the goal of more back ends without too much effort would be to use for example ML-RISC or `C--`[18].

Function calls As we have seen HiPE is much faster on tail-recursive calls to the same function (when caller and callee are the same). This is partly because this can be implemented as a one instruction branch. However all other types of calls take three instructions in the current implementation.

We could change ordinary calls to branches in those cases where a branch would be sufficient to reach its destination. This scheme would require some extra work in the code loader, since hot code loading in HiPE is implemented by back-patching the call site. This is because the back-patcher would have to be able to back-patch one instruction branches to three instruction jumps and vice versa.

8.3 Possible optimizations to investigate

There are some optimizations that we think could have a positive impact on HiPE but we are not as certain as with the ones described above. Still we think that these optimizations deserve further investigation and that they probably would have a positive effect. We will present these ideas here.

8.3.1 Optimizations of process communication

Message passing in ERLANG is done with the `send` primitive, where the destination is computed at run-time.

For each `send` statement in the code there is often one matching `receive` statement. In most cases (in the benchmarks we have studied and in the ERLANG shell) the receiving process is suspended with an empty mailbox in this receive statement.

This could be exploited by taking the code of the sender and the code of the receiver and optimize them together. One could view this as inlining over process boundaries as compared to over function boundaries.

This idea is presented in [14] and will be more thoroughly described in a forthcoming paper.

8.3.2 A global heap

Today `send` is implemented by copying the message from the senders heap to the heap of the receiver. Even though this has not showed up as a big cost in the programs we have examined here it might be a problem for programs with more concurrency. One thing that at least SCCT is suffering from is the copying of data to and from the ETS databases, which shows up as a lot of time spent in the database built-in functions.

One way of solving both these problems would be to just have one common heap for all processes. This would also decrease the total amount of allocated data, since all processes can share the same copy of a data structure. With this scheme GC becomes more complex, and there is a risk that the garbage collection time will increase. Robert Virding at Computer Science Lab at Ericsson has experimented with one unified global heap and describe two garbage collector algorithms [23, 22]. The algorithm that seemed most promising, with a simple generation scheme, had problems with large root sets, but no thorough evaluation of this algorithm has been performed. Today Mikael Pettersson is working on the design of an incremental generational garbage collector that seems to be able to solve these problems.

8.3.3 Compile-time GC

It is sometimes possible to find the last use of an object by doing liveness analysis on heap allocated data. This information could be used, for example, to store the object on the stack instead of on the heap and then reclaim the space as soon as it is no longer needed.

Another, maybe even more interesting technique, would be to reuse the heap space of the object. Take, for example, the built-in function `setelement/3` which copies the entire original tuple. If an analysis could tell us that the old tuple not is referred to after this call, we could safely mutate the tuple instead of copying it.

A problem is that the analysis would probably need the whole program to be really effective. Furthermore, special considerations would have to be taken with regards to `send` if this technique were to be combined with a global heap for all processes. But we can conceive scenarios where this technique could prove interesting in combination with optimizations of process communication as described in [14] and above.

This technique could be used for lists also, and we know that they are heavily used in ERLANG programs. One of the most called functions is `lists:reverse/1`. Provided that the original list is not referred to afterwards, `reverse` could reuse all *cons* cells from the original list.

A quick test shows that reversing a 2,000,000 elements long list takes about 5 s in emulated JAM code on our test machine if we force a garbage collection just before the reversal. (If we do not force the garbage collection it takes a very long time.)

If we compile the `reverse` function to native code it only takes about 290 ms. With reuse of *cons* cells it takes about 230 ms to reverse the list. The destructive reverse does not need to copy the elements in the list thus making it a little bit more effective.

The big gain comes from not needing any more heap space for the data; if we do not force the garbage collection then a normal reverse in native code takes about 1.5 s and a destructive reverse about 330 ms.

If we combine this technique with the global heap we need an escape analysis that can se that data escapes when sent as a message. It could be potentially more interesting for a system with one global heap to reuse heap space early since such a system need to keep the garbage collection times down. At the same time this technique might be hard to combine with a generational garbage collector, since we can get older objects that refers to younger objects.

8.3.4 Adaptive compilation

Many optimizations trade size for speed, larger more specialized code is faster than smaller more general code. The problem for very large programs is that not all the code can be specialized in this way, it would result in an enormous code volume. The usual solution is therefore to let the programmer tell the compiler what parts of the code are time critical, in order to get them heavily optimized. The problem is that it can be hard for the programmer to know exactly where the hot spot of the program is. One solution to that is to employ adaptive compilation. With adaptive compilation the run-time system monitors the execution of the programs and when a hot spot is identified that part of the code is recompiled with more optimizations.

ERLANG is very well suited for adaptive compilation since hot-code loading is a part of the language. In HiPE we can monitor the number of calls to each function, and even the number of times a basic block in native code is executed. We can also control the UltraSPARC low level performance counters directly from ERLANG code. This gives HiPE very good abilities to identify hot spots and their nature.

This together with HiPE's ability to compile and recompile ERLANG code to native code with several levels of optimizations, makes it possible to try several different schemes of adaptive compilation.

9 Conclusion

In this report we have shown how the HiPEgroup have instrumented three ERLANG systems in order to enable several types of measurements, including very low level measurements. We have also used this instrumentation to benchmark these ERLANG systems on real industrial programs, presented and analyzed the results of these measurements and outlined a plan for a faster ERLANG system.

9.1 Instrumentation of three Erlang run-time systems

By using the low level performance counters in UltraSPARC and other counters in the emulators and run-time systems we have made it possible to monitor the behavior of three ERLANG run-time systems very closely.

The low level performance counters in UltraSPARC has given us the tools to see the behavior of the pipeline and the memory hierarchy. There are some problems with the precision of these counters, since they can be somewhat overlapping, they can overwrap, and they measure all processes in the operating system. By doing repeated measurements in intervals of about one second, we get consistent results. The differences between the three ERLANG run-time systems are big enough to be significant. This makes it possible for us to monitor the low level behavior of these systems.

We can also see how much time the JAM system spends in built-in functions and in garbage collection. We can see this on the same scale as the low level measurements, and we can also measure the cache and pipeline behavior of the built-in functions and the garbage collector.

We can also control the low level counters directly from ERLANG code making it possible to write complex monitoring applications in ERLANG.

9.2 Real industrial benchmarks

It is unfortunately not common that researchers in functional programming languages have large industrial programs to test their optimizations on. Often the largest program is the optimizing compiler it self, an interesting program but maybe not representative.

There are several reasons for this. For a new language there are often no larger programs at all. If there are some large programs written they are probably not open source or available. And if they are available it is often much harder to do good measurements on a large program than on a simple toy benchmark.

When we have tried to get our hands on "real" programs we have also often failed, even though the ERLANG community has been very forthcoming and interested in our project. The problem has been that since the programs are real products, the programmers that want to share their programs with us have not been allowed to do this by their superiors. Even if they have been allowed to give us the source code they have been busy with the next version and have not been able to tell us what the program does and how it is used and what would be interesting to measure.

Fortunately we have been able to get some programs that are well suited for performance measurements, because they have been designed to test the performance of some aspect of a real program. Since the ERLANG community is very open and commited to their language we hope that we will be able to get more industrial size benchmarks in the future.

The ones that we have gotten have shown us that these larger programs behave quite different from toy benchmarks such as *length*. This is not very surprising.

We think that large real world applications are important for the evaluation of research compilers, and in this report we have presented such an evaluation.

9.3 Analysis of the results

As we have seen the code volume for the native code is a lot larger than for JAM, and one could imagine that this would be a problem on large programs. We have seen that HiPE spend about 20% (on SCCT) of its time stalling because of instruction cache misses but in absolute numbers the time spent waiting on the instruction cache is small.

If we look at all stalls together both JAM and HiPE spends about 20% of their time stalling. JAM has troubles both with load stalls and mispredictions. The load stalls might come from JAM being a stack machine and from the loading of JAM instructions. The mispredictions might be from the threaded implementation of JAM and from the fact that the outcome of tests in JAM are unpredictable.

Our conclusion is that a threaded implementation and a stack-based virtual machine are not a perfect match with a modern superscalar RISC architecture such as UltraSPARC.

We can also conclude that even simple and naive native compilation without any instruction scheduling is effective even for large programs that do not fit in the on-chip instruction cache.

For SCCT the performance for BEAM is not far from the performance of

HiPE. It is hard for any one compiler to get a good absolute performance gain on SCCT, since so much time is spent in built-in functions.

We think that HiPE faces three problems:

- HiPE uses the same front end as JAM
- HiPE only compiles one function at a time.
- HiPE can not improve on the time spent in built-in functions.

In order to tackle all three problems and get a significant increase in performance, an attack on a broad front is needed.

From the results we have seen we have been able to identify possible optimizations that we think will be important for ERLANG in the future. We have also presented some interesting venues for further investigation.

Index

References

[1] Hassan Ait-Kaci. *Warren's Abstract Machine.* The MIT Press, Cambridge, MA, 1991.

[2] J. L. Armstrong, M. C. Williams, C. Wikström, and S. R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 2nd edition edition, 1995.

[3] Joe Armstrong. The development of Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-97)*, volume 32,8 of *ACM SIGPLAN Notices*, pages 196–203, New York, June 9–11 1997. ACM Press.

[4] Joe Armstrong and Robert Virding. One-pass real-time generational mark-sweep garbage collection. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[5] The Group Eddie. Eddie. `http://wwweddie.serc.rmit.edu.au/`.

[6] Telecom AB Ericsson. Axd 301 high-performance atm switching system. Technical Report EN/LZT 108 662 R2, Ericsson Telecom AB, `http://www.ericsson.se/datacom/products/axd301/more_axd_301.shtml`, 1998.

[7] J Halén, R Karlsson, and M Nilsson. Performance measurements of threads in java and processes in erlang. Technical Report ETX/DN/SU-98:024, Ericsson, http://www.ericsson.se/cslab/ joe/du98024.html, November 1998.

[8] Bogumil Hausman. Turbo Erlang. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, page 662, Vancouver, Canada, 1993. The MIT Press.

[9] Bogumil Hausman. Turbo erlang: Approaching the speed of C. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994.

[10] Bogumil Hausman. Hybrid implementation techniques in Erlang BEAM. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 816–816, Cambridge, June 13–18 1995. MIT Press.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[12] Erik Johansson and Christer Jonsson. Native code compilation for Erlang. Uppsala master thesis in computer science 100, Uppsala University, 1996.

[13] Erik Johansson, Christer Jonsson, Thomas Lindgren, Johan Bevemyr, and Håkan Millroth. A pragmatic approach to compilation of Erlang. Technical Report UPMAIL No. 136, Uppsala University, February 1997.

[14] Erik Johansson and Sven-Olof Nyström. Optimization of asynchronous communication in concurrent functional languages. `http://www.csd.uu.se/~happi/publications/proposal.ps`, 1998.

[15] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[16] Thomas Lindgren. Module merging: aggressive optimization and code replacement in highly available systems. Technical Report UPMAIL Technical Report No. 134, Uppsala University, March 1998.

[17] Sun Microelectronics. The UltraSPARC processor. Technical Report Technology White Paper, Sun Microsystems, `http://www.sun.com/microelectronics/whitepapers/`.

[18] S. Peyton Jones, T. Nordin, and D. Oliva. C–: A portable assembly language. *Lecture Notes in Computer Science*, 1467:1–??, 1998.

[19] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Computer Science. Prentice-Hall, 1987.

[20] R. Radhakrishnan and L. John. Execution characteristics of object oriented programs on the ultrasparc-ii. In *Proceedings of the 5th International Conference on High Performance Computing*, December 1998.

[21] Michael Rumsewicz. Web servers for commercial environments: The imperatives and the solution. Technical Report EWP-001, March 1999.

[22] R. Virding. A garbage collector for the concurrent real-time language erlang. *Lecture Notes in Computer Science*, 986:343–??, 1995.

[23] Robert Virding. A garbage collector for the concurrent real-time language Erlang. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[24] M. C. Williams, J. L. Armstrong, S. R. Virding, and B. O. Däcker. Implementing a Functional Language for Highly Parallel Real Time Applications. In *8th SETSS, Florence, Italy, March 30 – April 1*, 1990.