Performance Measurements and Process Optimization for Erlang

Erik Johansson

Uppsala University
Information Technology
Computing Science Department

Thesis for the Degree of Licentiate of Philosophy

UPPSALA 1999
Performance Measurements and Process Optimization for Erlang

ERIK JOHANSSON

A Dissertation submitted in partial fulfilment of the requirements for the Degree of Licentiate of Philosophy at Computing Science Department, Information Technology, Uppsala University.
Abstract


In this thesis I present an evaluation of the Erlang native code compiler called HiPE, and a method for performing inter-process optimization in Erlang.

I compare three Erlang systems that use the same garbage collector and the same built-in functions, making it possible to compare the impact of different front ends and execution models.

I have used large real-world applications as benchmarks when comparing these systems. To gather information about the performance I have used the low-level performance counters available on the UltraSPARC processor.

The measurements show that the speedup for the native code compiler HiPE over a byte code emulator with the same front end varies significantly depending on the executed program, from 16 times for a small sequential benchmark to 1.6 times for a large concurrent program that spends much of the execution time in the built-in functions of the run-time system.

I also present a method for merging code from communicating Erlang processes. I have implemented a prototype inter-process optimizer (Hippo), which uses profiling to detect how and when processes are communicating.

I describe the design of an Erlang process communication profiler. For the programs I have studied the communication behavior is consistent enough to make inter-process optimizations possible.

*Erik Johansson, Computing Science Department, Information Technology, Uppsala University, Box 311, SE–751 05 Uppsala, Sweden.*

© Erik Johansson 1999

ISSN 0283–359X
To my Parents
ACKNOWLEDGMENTS

I would first of all like to thank my hands on supervisor Sven-Olof Nyström who has helped me enormously, not only with the writing of this thesis but also with the analysis of the results and with the design of Hippo.

My supervisor Håkan Millroth who started the HiPE group and introduced me to ERLANG deserves a large thank you.

I would also like to thank my previous hands on supervisor Thomas Lindgren for introducing me to the UltraSPARC performance counters and for his work on the HiPE compiler (especially the basic block profiling), and for preparing the SCCT benchmark for execution in HiPE.

A very special thank you goes to Christer Jonsson with whom I for a long time shared an office, agony, and a HiPE compiler. Christer not only implemented the optimizations in the current version of the HiPE compiler he also managed to make the SCCT benchmark executing in our instrumented BEAM system.

I would also like to thank Ulf Wiger the chief designer of AXD 301 and everyone else at “taket” for letting us use the SCCT benchmark and publish the results, and Jon Larsson and Torbjörn Törnvist at The Software Engineering Research Centre of The Royal Melbourne Institute of Technology for providing the HTTP-server from “Eddie”.

This research was funded by ASTEC, and I am very grateful to ASTEC and Ericsson for funding my research.

Richard Carlsson and Per Mildner have read early drafts of this thesis and helped me with comments and the English language, thank you both.

I would like to thank my parents for supporting me, sometimes even financially, this thesis is for you.

Last but not least I would like to thank Cilla who have made these last two years of my life so wonderful.
CONTENTS

1 INTRODUCTION

2 BACKGROUND

2.1 ERLANG 6
2.1.1 Basic properties of ERLANG 6
2.1.2 Modules 8
2.1.3 Concurrency 10
2.1.4 Exceptions 13
2.1.5 Meta call 13
2.1.6 Memory management 13
2.1.7 Distributed ERLANG 13
2.1.8 Built-in functions and libraries 14

2.2 Three ERLANG run-time systems 14
2.2.1 Common framework 15
2.2.2 JAM 18
2.2.3 BEAM 20
2.2.4 HIPE 25
2.2.5 Other ERLANG implementations 28

2.3 Experimental setup 28
2.3.1 Hardware 29
2.3.2 Performance measurement on UltraSPARC 34
2.3.3 Instrumentation of HiPE 36

3 MEASUREMENTS 38

3.1 Performance on a sequential benchmark 39
3.1.1 Generated code 40
3.1.2 Speedup 46
3.1.3 Instructions and clock cycles .................................. 47
3.1.4 Pipeline stalls .................................................. 48
3.1.5 Different types of calls ........................................ 50
3.1.6 Conclusion ...................................................... 52
3.2 The Eddie benchmark ............................................... 53
  3.2.1 Instructions and clock cycles ................................. 54
  3.2.2 Pipeline stalls ................................................ 54
  3.2.3 Different types of calls ..................................... 56
  3.2.4 built-in functions .......................................... 56
  3.2.5 Conclusion ..................................................... 57
3.3 SCCT, the AXD 301 benchmark .................................... 57
  3.3.1 Instructions and clock cycles ................................. 59
  3.3.2 Different types of calls ..................................... 60
  3.3.3 The impact of the memory hierarchy ....................... 61
  3.3.4 Concurrency .................................................. 64
  3.3.5 Built-in functions .......................................... 65
  3.3.6 Conclusion ..................................................... 68
3.4 Related Work .................................................... 69
3.5 Conclusion ........................................................ 71

4 HIPPO: AN INTER-PROCESS OPTIMIZER .............................. 75
4.1 Profiling .......................................................... 76
  4.1.1 Why not use analysis? ...................................... 77
  4.1.2 Description of the profiler ................................ 78
  4.1.3 Results ....................................................... 79
4.2 The transformation ................................................ 80
  4.2.1 The merged code ............................................ 82
  4.2.2 Further Considerations .................................... 85
  4.2.3 Return messages ............................................ 85
  4.2.4 An example .................................................. 87
4.3 Related Work ..................................................... 92
4.4 Conclusion ........................................................ 93
  4.4.1 Potential gains .............................................. 93
  4.4.2 Reduced message passing ................................. 93
  4.4.3 Short-circuited switches on messages .................... 93
  4.4.4 Reduced context switching ............................... 94
  4.4.5 Enabling of further optimizations ....................... 94
  4.4.6 Results ....................................................... 94
## Conclusion

5.1 Measurements ............................................. 97
5.2 Inter-process optimization .............................. 100

Index ......................................................... 102

References .................................................. 104
LIST OF FIGURES

2.1 ERLANG processes and other data structures in the run-time system. ............... 16
2.2 The execution loop of the ERLANG run-time system. 17
2.3 Representation of the term \{42, [foo, bar]\}. 18
2.4 The stack after a call. 20
2.5 Intermediate representations in HiPE. 26
2.6 Memory and pipeline architecture on UltraSPARC. 31

3.1 Performance ratio for the benchmark Length. 47
3.2 The number of executed instructions (in millions) and execution time (in millions of clock cycles) for the benchmark Length. 48
3.3 Total execution time and pipeline stalls for the benchmark Length. 49
3.4 Speedup ratio of execution times for benchmarks with different types of calls in HiPE compared to JAM. 51
3.5 Stalls in comparison to total execution time for the Eddie HTTP parser benchmark. 55
3.6 The total running time for SCCT. 60
3.7 Pipeline stalls in relation to total execution time. 62
3.8 Pipeline stalls in percent of total execution time. . . . 64
3.9 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. . . . . . . . . . . . . . . . . . . . . 65

4.1 The lattice of possible Dest values. . . . . . . . . . 78
4.2 Before the merging, function $f$ is executed by $\alpha$ and function $g$ is executed by $\beta$. After the merging, $f'$ is executed by $\alpha$. . . . . . . . . . . . . . . . . . . . . 81
List of Tables

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

3.2 Number of different calls in the HTTP parser

3.3 Time that JAM spends in built-in functions when running the HTTP parser

3.4 Number of different calls in SCCT

3.5 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

3.6 Execution times for the benchmarks. The Mc column is rounded average execution times in million of cycles. The Mi column is the average number of millions executed instructions. The CPI column is the number of cycles per instruction (the lower the better)

3.7 Relative speedups for BEAM and HiPE as compared to JAM. The speedups are calculated from the total execution times in clock cycles
3.8 Percentage of the execution time spent in the most common types of pipeline stalls. The columns %L shows the percentage load stalls. The columns %M shows the percentage Misprediction stalls. The columns %IC shows the percentage of instruction cache stalls. And the T columns shows the total percentage. All numbers are based on the number of executed machine cycles.

73
INTRODUCTION

In this thesis I present an evaluation of the ERLANG native code compiler called HiPE, and a method for performing inter-process optimization in ERLANG.

ERLANG is a concurrent functional programming language developed by Ericsson. HiPE is a SPARC native code compiler for ERLANG developed by the High Performance ERLANG group at Uppsala University. Hippo is a prototype inter-process optimizer for ERLANG. With inter-process optimization I refer to optimization of code from two processes that communicate with respect to the information communicated.

ERLANG has proved itself in several projects both inside and, especially in later years, outside Ericsson. With ERLANG, huge fault-tolerant systems have been developed on time. One reason for the slow acceptance and spread of functional programming languages (FPL) in industry has been that most FPL implementations have been considered slow. The ERLANG implementations provided by Ericsson have proved themselves fast enough for many projects, but still there are worries that an ERLANG user pays for the high-level abstractions of the language by slower execution.

In the HiPE group we want to achieve faster execution despite the presence of these high-level abstractions. As a first step toward this
goal we have implemented a native code compiler for ERLANG and instrumented three ERLANG run-time systems in order to identify bottlenecks in the implementations.

It is possible that we in the future can implement a compiler that is aware of the meaning and intention of these abstractions and that such a compiler can produce code that is even faster than handwritten low-level implementations.

In this thesis I compare two ERLANG emulators with a native code compiler. All three implementations share large parts of the run-time system and can execute programs compiled from the same source code, making it possible to compare how these parts of the designs that differ affect the performance and the execution behavior. I have used low-level performance counters in the UltraSPARC processor to see how implementation decisions effects the cache behavior and the throughput of instructions in the processor.

I also present a method for merging code from communicating ERLANG processes. I have implemented a prototype inter-process optimizer (Hippo), which uses profiling to detect how and when processes are communicating. Hippo and the optimizations it performs is also described in this thesis.

The contributions of my research are:

- I have implemented instrumentation tools for use in ERLANG. Christer Jonsson and Thomas Lindgren have also contributed to these tools.

- I have compared three ERLANG systems that share large parts of the run-time system. This makes it possible to clearly see the impact on performance of the implementation differences. These systems use the same garbage collector, the same built-in functions, and basically the same data representation. Two of the systems use the same front end, making it possible to
see the impact of different execution models (byte code interpreted and native code execution). The third system, which has an execution model that lies between byte code interpretation and native code execution, uses a different front end making it possible to see the impact of the front end on performance.

- I have used large real-world benchmarks instead of tiny toy benchmarks in order to see how the differences will effect real users.

- I have made a thorough investigation of the implications of the designs on low-level performance aspects such as cache effects and number of instructions executed per machine clock cycle.

- I present the idea of inter-process optimization in ERLANG and a solution using code-merging together with a prototype implementation.

- I present the design and implementation of a profiling-system for message passing that can determine when inter-process optimization can be performed.

- I show that in many cases inter-process optimization can be performed.

- I also present a simple method for passing information, which the intermediate code can not express, between compilation stages. This is done by assigning information to new synthetic temporaries which later are removed by the constant propagator.

The measurements have had a two-fold aim. The first has been to examine the performance of the code generated by the HiPE compiler in order to identify possible bottlenecks. The second aim has been to compare emulated code with native code when executing very large
programs, and to answer the question whether native code would suffer from instruction cache misses.

In order to achieve these goals I first present how the run-time systems behave on a small program with only sequential code. This way we can see how much faster HiPE can be and compare this with the results on very large programs. In order to see the cache behavior of the different implementations I have used the low-level performance instrumentation available on the UltraSPARC.

The goal with the inter-process optimizer has been to examine whether ERLANG programs show a communication behavior that is sufficiently consistent to allow inter-process optimizations and in that case present a method for performing inter-process optimization.

The first goal is reached by the implementation of a process communication profiler and the application of this profiler to the benchmarks. To reach the second goal I have implemented a prototype inter-process optimizer. This implementation shows that the method is feasible.

Chapter 2 presents background material needed for the understanding of the rest of the thesis. In this chapter I first present the parts of ERLANG that are interesting and necessary for the understanding. Then I present the three ERLANG run-time systems I have used, their common parts, and their differences. Finally I also present the experimental setup for the measurements, including a description of the UltraSPARC design and a description of how the ERLANG run-time systems are extended with performance instrumentation.

In the following chapter (Chapter 3) I present the benchmarks with results from each run-time system. Each benchmark is presented in its own section, beginning with a sequential benchmark followed by an http parser and finally the SCCT benchmark, which is a part of
Ericsson’s ATM switch AXD301. The chapter ends with a survey of related work.

Chapter 4 presents The HiPE inter-process profiler and optimizer (Hippo), the design of the profiler followed by the code transformations used to merge code from two processes. There is also a short survey of related work and some conclusions drawn from the implementation of the prototype are presented.

Finally in Chapter 5 I present some conclusions drawn from the entire thesis.
BACKGROUND

In Section 2.1 of this chapter I present the parts of ERLANG that are interesting and necessary for the understanding of the rest of the thesis. Then (in Section 2.2) I present the three ERLANG run-time systems I have used, their common parts, and their differences. Finally in Section 2.3 I also present the experimental setup for the measurements, including a description of the UltraSPARC design and a description of how the ERLANG run-time systems are extended with performance instrumentation.

2.1 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.

2.1.1 Basic properties of Erlang

All variables and data structures in ERLANG are immutable. Syntactically ERLANG variables begins with a capital as the variables Rest, AccLen, and Length in Example 1.
Example 1 (Two simple functions in Erlang)

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

% The function \texttt{len/1} calculates the length of a list.
\texttt{len(List) \rightarrow len(List, 0).} % By calling \texttt{len/2} with the list and zero.

% The function \texttt{len/2} calculates the length of a list.
\texttt{len([\ldots|Rest], AccLen) \rightarrow len(Rest, AccLen+1);}
\texttt{len([], Length) \rightarrow Length.}

\texttt{ERLANG} has no iteration constructs, but loops can be constructed by recursion. Preferably by \textit{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 \texttt{len/2} is tail-recursive.

\texttt{ERLANG} is dynamically typed and there is no explicit way for the programmer to specify new datatypes\textsuperscript{1}. But there are implicit ways to construct complex data structures from the datatypes present in the language. The simplest datatype in \texttt{ERLANG} is the \textit{atom}, two atoms are identical if and only if they have the same name. There are three different types of numbers in \texttt{ERLANG}: \textit{fixnums}, \textit{floats}, and arbitrary precision numbers (\textit{bignums}). \texttt{ERLANG} has some other simple datatypes, such as process identifiers (\textit{PIDs}), \textit{references}, and \textit{ports}. There is also a special datatype, called a \textit{binary}, for (large) sequences of bits, which is often used for incoming and outgoing communication. The implementation of binaries has to be both time- and space-

\textsuperscript{1}\texttt{ERLANG} does have a definable data structured called a \textit{record}, but records can be converted to tuples by a preprocessor step.
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 and tuples.
A list is either empty (\([\square]\)\), called nil, or a cons of some data structure
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 \([\square]\) and the
fixnum 1.

ERLANG supports pattern matching, where patterns of datatype con-
structors can be used to distinguish between different cases. An un-
bound 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
\(\[\square]\) only matches the empty list.

2.1.2 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 (with the export
directive) from the module defining them. Within the module, how-
ever, all functions are visible. Example 2 shows a complete ERLANG
module.
2.1. ERLANG

A call to a function in another module is called a *remote call*, 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)**

```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([], AccLen) ->
    AccLen;
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 one module at a time. During the transition, both old code and new code can coexist. After a new code for a module has been 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.
2.1.3 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([f/0, g/0]).

% Code for the sender Alpha
f() ->
    Beta_PID = spawn(process_comm, g, []),
    Beta_PID ! {ping, self()}, % send a message.
    receive % Wait for a message...
        pong -> true; % We hope to get a pong back.
        _ -> false
    end.

% Code for the receiver Beta
g() ->
    receive
        {ping, Alpha_PID} -> % Receive a ping
            Alpha_PID ! pong; % ... and respond
        _ ->
            throw(unknown_message) % We don’t expect this
            after 10000 ->
                throw({time_out}) % If we don’t get any...
        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 g/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:

\[ \text{Beta_PID ! \{ping, self()\}} \]

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

The built-in function \( \text{self}() \) returns the process identifier of the current process.

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

\[ \text{receive Message -> ... end} \]

This expression will check the mailbox for \textit{any} messages and return the first message in the mailbox. If the mailbox is empty the process will be \textit{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, 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 \texttt{'}~\texttt{'} as in Example 3 to get rid of unwanted messages.

The \texttt{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 \texttt{timeout} in the \texttt{receive} 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.

When a process α, sends a message to a process β, it often expects an answer. In ERLANG this is expressed as follows. In the sender the message is followed by a receive:

```
...
Beta ! {question, self()},
receive
  {Beta, Answer} -> ...;
... -> ...
end,
...
```

The sender includes its own process identifier (with self()) in the message. The receiver β is expected to respond by sending a message back to the process mentioned in the message:

```
...
receive
  {question, From} ->
    From ! {self(), 42};
... -> ...
end,
...
```

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
2.1. ERLANG

is used for communication with the host operating system and for interaction with programs written in other languages.

2.1.4 Exceptions

Error recovery is an important part of ERLANG and all run-time errors are trappable by means of a catch. A catch works like a handle in ML and all exceptions generated in a “caught” 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.

2.1.5 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.

2.1.6 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.

2.1.7 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.
CHAPTER 2. BACKGROUND

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.

2.1.8 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\(^2\).

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 are called ETS-tables. An ETS-table could either be private to a process or public, that is, accessible to all processes that have the unique reference to that ETS-table. The data in the databases are tuples where the first element in the tuple is the key.

2.2 THREE Erlang RUN-TIME SYSTEMS

I have looked at three different ERLANG implementations: JAM, BEAM, and HiPE\(^3\). 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 chapter I describe some aspects of these implementations and the similarities and differences between the systems.

\(^2\)A specific telecom library provided by OTP is also available for ERLANG. Information about OTP (and ERLANG) can be found at \url{http://www.erlang.se}

\(^3\)I will use the names JAM, BEAM, and HiPE to refer both to the run-time system and the compiler. I might also say that a benchmark was run on JAM when I 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.
2.2. THREE ERLANG RUN-TIME 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 tagging scheme. This makes it possible to compare how the differences affects performance. That is, the back ends and the emulators.

2.2.1 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 I describe how these systems handle processes and scheduling.

Processes  From the view of the operating system, the ERLANG run-time system is merely an application with ERLANG 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 [11].

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 2.1. This means that data has to be copied to and from the process heap and the database when a process performs a database access.
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.

**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 2.1. 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 2.2.
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 suspended 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.
CHAPTER 2. BACKGROUND

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.

2.2.2 JAM

JAM [30, 3] is a stack-based byte code emulator for ERLANG. The version of JAM that I have used is a modified version of JAM version 4.5.3. To this version Christer Jonsson and I have added support for measurements and support for the integration with HiPE. The development of JAM has not stood still while the HiPE group has been developing HiPE, the latest version of ERLANG provided by Ericsson is at the time of writing 4.8.

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

![Figure 2.3: Representation of the term \{42, [foo, bar]\}.](image)

Data representation 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
2.2. **THREE ERLANG RUN-TIME SYSTEMS**

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 2.3.

**Emulator** The C-compiler gec 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.

**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 2.4). 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 to the same stack position as the first argument.

**Tweaking** When I did my first measurements I 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 I 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 I have not noticed any ill effects because of it.

2.2.3 BEAM

BEAM [12, 13, 14] is a threaded emulator for an abstract register machine. The design of BEAM is influenced by the Warren Abstract Machine (WAM)[2], which was designed for the language Prolog and used in many Prolog implementations.

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.

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.

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.

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 [24] is not implemented.

Let us look at an example. In the simple pattern matching example (Example 4) there are four different patterns that are mutually exclusive, which means that we could test them in any order we want. The best would be to group all clauses containing a tuple together (in rough pseudo code):
• 1. Is \( \text{arg0} \) a tuple then 2a else 2b.

• 2a. case element(1, \( \text{arg0} \)) of

  - a: return(element(2, \( \text{arg0} \))
  - b: return(element(2, \( \text{arg0} \))
  - c: return(element(2, \( \text{arg0} \))
  - default: fail.

• 2b. is \( \text{arg0} \) the atom \( c \) then return(\( c \)) else fail.

**Example 4 (Erlang code for matching a simple pattern)**

\[
\begin{align*}
test([a, V]) & \to V; \\
test([b, V]) & \to V; \\
test(c) & \to c; \\
test([d, V]) & \to V.
\end{align*}
\]

Unfortunately neither JAM nor BEAM does this. JAM just tests each clause by itself until a match is found (Example 5). BEAM groups similar patterns together as long as no "non-similar" patterns come in between (Example 6). The method of BEAM is of course more effective than that of JAM. Since ERLANG functions tend to rely heavily on pattern matching it is important to have a good pattern matcher.

**Example 5 (JAM code for Example 4)**

\[
\begin{align*}
\text{info(pattern,test,1)} & \quad \text{This is the function pattern: test/1} \\
\text{try_me_else(22)} & \quad \text{Set up label 22 as the fail point} \\
\text{arg(0)} & \quad \text{Get the first argument} \\
\text{unpKTuple(2)} & \quad \text{Is it a tuple of arity 2? (no \( \to \) 22)} \\
\text{get(a)} & \quad \text{Is the first element the atom a?} \\
\text{storeVar([0,[var,0]])} & \quad \text{Bind variabel V to element 2} \\
\text{commit} & \quad \text{Remove the fail point 22} \\
\text{pushVar([0,[var,0]])} & \quad \text{Get variable V}
\end{align*}
\]
2.2. THREE ERLANG RUN-TIME SYSTEMS

\[
\begin{array}{l}
\text{ret} \quad ; \text{Return (V)} \\
\text{try\_me\_else(23)} \quad ; \text{Label 22 (the first fail point)} \\
\text{alloc(1)} \quad ; \text{Set up label 23 as the fail point} \\
\text{arg(0)} \quad ; \text{Make room for a local variable (V)} \\
\text{unpkTuple(2)} \quad ; \text{Get the first argument} \\
\text{get(b)} \quad ; \text{Is it a tuple of arity 2? (no -> 23)} \\
\text{storeVar}\{0,\{\text{var,0}\}\} \quad ; \text{Check the first element of the atom b?} \\
\text{commit} \quad ; \text{Bind variable V to element 2} \\
\text{pushVar}\{0,\{\text{var,0}\}\} \quad ; \text{Remove the fail point 23} \\
\text{ret} \quad ; \text{Get variable V} \\
\text{Return (V)}
\end{array}
\]

\[
\begin{array}{l}
\text{try\_me\_else(23)} \quad ; \text{Return (V)} \\
\text{alloc(1)} \quad ; \text{Set up label 23 as the fail point} \\
\text{arg(0)} \quad ; \text{Make room for a local variable (V)} \\
\text{unpkTuple(2)} \quad ; \text{Get the first argument} \\
\text{get(b)} \quad ; \text{Is it a tuple of arity 2? (no -> 23)} \\
\text{storeVar}\{0,\{\text{var,0}\}\} \quad ; \text{Check the first element of the atom b?} \\
\text{commit} \quad ; \text{Bind variable V to element 2} \\
\text{pushVar}\{0,\{\text{var,0}\}\} \quad ; \text{Remove the fail point 23} \\
\text{ret} \quad ; \text{Get variable V} \\
\text{Return (V)}
\end{array}
\]

\[
\begin{array}{l}
\text{try\_me\_else(23)} \quad ; \text{Label 22 (the first fail point)} \\
\text{alloc(1)} \quad ; \text{Set up label 23 as the fail point} \\
\text{arg(0)} \quad ; \text{Make room for a local variable (V)} \\
\text{unpkTuple(2)} \quad ; \text{Get the first argument} \\
\text{get(b)} \quad ; \text{Is it a tuple of arity 2? (no -> 23)} \\
\text{storeVar}\{0,\{\text{var,0}\}\} \quad ; \text{Check the first element of the atom b?} \\
\text{commit} \quad ; \text{Bind variable V to element 2} \\
\text{pushVar}\{0,\{\text{var,0}\}\} \quad ; \text{Remove the fail point 23} \\
\text{ret} \quad ; \text{Get variable V} \\
\text{Return (V)}
\end{array}
\]

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 5, 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 6 (BEAM code for Example 4)

44:
    func_info('pattern','test',1)
1:    if not is_tuple(x(0)) then 6
       if not arity(x(0)) == 2 then 6
       x(1) := get_tuple_element(x(0), 0)
       if not x(1) == 'a' then 2
       x(2) := get_tuple_element(x(0), 1)
       x(0) := x(2)
       return
2:    if not x(1) == 'b' then 3
       x(2) := get_tuple_element(x(0), 1)
       x(0) := x(2)
       return
3:
6:    if not x(0) == 'c' then 4
       x(0) := 'c'
       return
4:    if not is_tuple(x(0)) then 5
       if not arity(x(0)) == 2 then 5
       x(1) := get_tuple_element(x(0), 0)
       if not 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. 4

As can be seen in Example 6, 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

4The instruction if not ... 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 if not ... else instruction.
must, again, check if the argument is a tuple and load the first element if it is.

2.2.4 HiPE

HiPE is the name of our project, the High Performance Erlang project at Uppsala University. HiPE is developed on top of version 4.5.3 of Ericsson’s JAM[18]. The main goal has been to implement a native code compiler for the SPARC architecture.

Integration with JAM In the HiPE run-time system both emulated JAM code and native compiled SPARC code can be executed, even within the same Erlang process. To make it possible for emulated and native code to share data on the same heap HiPE uses exactly same tagging scheme as JAM.

However the JAM calling convention is not used in native code, 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.

HiPE uses 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-frames6 that have links between the stacks 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 HiPE uses a higher number of reductions when executing native code then what is used in the emulator.

---

5 A previous Erlang implementation (JERICO) [17] 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.
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 my immediate needs.\textsuperscript{7}

\textit{Compilation} The HiPE compiler is for the moment a run-time compiler only. That is, it can only compile \texttt{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 \texttt{ERLANG} terms, but this is rather clumsy.)

\begin{figure}
\centering
\includegraphics[width=\textwidth]{image.png}
\caption{Intermediate representations in HiPE.}
\end{figure}

\textsuperscript{7}The differences in the sizes of time-slices for native and emulated code does not effect the benchmarks since all executed code is compiled to native code.
2.2. THREE ERLANG RUN-TIME SYSTEMS

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 2.5 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 HiPE performs 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

\(^{8}\)Some instructions such as floating point instructions are not implemented as HiPE does not use them. Some extra instructions such as load_atom are added to make life easier.
be back-patched when the implementation of the callee function is changed. This way HiPE 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.

2.2.5 Other Erlang implementations

There exists some other Erlang implementations than JAM, BEAM, and HiPE. Robert Virding [29, 4] has developed two versions of an emulator called VEE in order to experiment with one unified heap that is shared between all processes.

Mark Feely et al [10] have developed an Erlang to Scheme compiler, ETOS, that has shown promising results on sequential benchmarks.

The performance of these systems are outside the scope of this thesis, mainly because neither of these systems have been fully developed yet, making it impossible to execute the large benchmarks that uses many built-in functions.

2.3 EXPERIMENTAL SETUP

I have conducted my 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 the term cy-
2.3. EXPERIMENTAL SETUP

cycles per instruction (CPI) is used 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.

The HiPE group has extended the HiPE system in order to gather this low level information in an easy and efficient manner, and also to enable other kinds of performance profiling.

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

2.3.1 Hardware

I run the benchmarks on a 143 MHz single processor Sun Microsystem Ultra 1 Model 140 with 128 MB of memory running Solaris 2.6. I use this system because it is the fastest single processor UltraSparc system at our department. (In the section about performance measurement on UltraSparc (Section 2.3.2) I 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.

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 can be seen in Figure 2.6.

The external cache can handle one access per cycle. These accesses are pipelined and after 3 cycles 16 bytes are returned.
On-chip data cache:  On-chip instruction cache:

| +------------------ | +------------------ |
| | | ^ | ^ | SET 0 | | 256 |
| | | | | 512 | | ^ lines |
| | | | | lines | | ^ |
| | | | | SET 1 | | 256 |
| | | | | ^ v | | ^ lines |
| +------------------ | +------------------ |
| ^ ^ | ^ ^ | 32 bytes line | 32 bytes line |
| ^ ^ | ^ ^ | 16 bytes 16 bytes |
| (8 instructions) |

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.

**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 2.6: 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 2.6, 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 2.3.1)

**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.

_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
2.3. EXPERIMENTAL SETUP

A cache line, only one, two, or three instructions are fetched instead of four.

Up to 12 prefetched instructions are stored in the instruction buffer, 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.

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[22].

This information is used as described in [15]: 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.

2.3.2 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 \((\text{PIC0 and PIC1})\) of the PIC register are specialized to measure different aspects.

The HiPE system 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.
2.3. EXPERIMENTAL SETUP

- 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.

Potential problems  There is no way to clear the PIC register or to control when the register will wrap. 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 a single processor machine has to be used or the process has to be bound 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-bits register but Solaris 2.6 is not a full 64-bits 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 I do not run any other 64-bits programs (except for the OS) when I do my 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 I make several runs and they all produce similar measurements, I can conclude that the measurements have not been drastically affected by this problem. To handle this in practice, I 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 it measure the behavior of all processes running concurrently on the CPU, and not only the benchmark process. This I deal with by running on an unloaded machine. Still I 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.

### 2.3.3 Instrumentation of HiPE

In our run-time system I 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.
2.3. EXPERIMENTAL SETUP

An ERLANG program can, by a call to a BIF, read the value in the TICK register that counts cycles.

Counters Each time a function is called, be that locally, remote, or by a meta-call (apply), a counter for that function isIncremented. HiPE 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.

The counters in the emulator can count how many times each JAM instruction is executed. The HiPE compiler can take an option flag that turns on profiling of each basic block. That way we can see how many times each basic block is executed and consequently how many times each native code instruction is executed.

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.
Chapter 3

Measurements

Here I present the benchmarks I have used for the performance measurement.

I have had two goals with these measurements. The first goal has been to examine the performance of the code generated by the HiPE compiler in order to identify possible bottlenecks. The second goal has been to compare emulated code with native code when executing very large programs, and to answer the question whether native code would suffer from instruction cache misses.

In order to reach these goals I first present how the three ERLANG systems described earlier behave on a very small sequential program. This way we can see how much faster HiPE can be, then we can compare this with the results on very large programs.

In order to see the cache behavior of the different implementations I have used the low-level performance instrumentation available on the UltraSPARC.

I have come to the conclusion that the HiPE system is capable of executing small benchmarks of sequential code much faster than the two emulators. And although the native code suffers from instruction cache misses on the larger programs, the two emulators also have problems with the hardware because of misprediction and load stalls. The main reason that HiPE does not get the same speedup on the
larger programs seems to be that much of the execution time is spent in built-in functions.

In Sections 3.1 I present the sequential benchmark, analyses the results and draw some conclusions. In Section 3.2 and Section 3.3 I present the HTTP parser from Eddie and SCCT, a time-critical part of the AXD 301 ATM switch, in the same way. The chapter ends with a survey of related work in Section 3.4 followed by a conclusion in Section 3.5.

### 3.1 PERFORMANCE ON A SEQUENTIAL BENCHMARK

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 I run this benchmark 20 times (with each system) and compute the average.

The systems I have looked at are HiPE 0.2, JAM 4.5.3 (modified), and BEAM 4.3. I 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 the changes have affected the emulator and to see how a more modern emulator behaves. The modifications to 4.5.3 have 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([], L) ->
    len(X, L+1);
len([X], L) ->
    L.

3.1.1 Generated code

I will present generated code from the three different compilers in order to give you a feeling of how they differ. I will not present the complete length benchmark but just the function len/2 which constitutes the inner loop of the benchmark.

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 failpoint 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 (popped). Then the tail of the list is saved in variable 0 \((X)\). The failpoint is now removed (with 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 continues 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)**

```plaintext
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
```

In the emulator the dispatch of each JAM-instruction requires at least 3 instructions: At least one load, there might be additional loads if the JAM instruction takes arguments. Then another load is required
to get the address of the code of the instruction. The emulator then jumps to this address to execute the JAM instruction. This means that in addition to the actual code that does the work there are at least 39 SPARC instruction in the inner loop of the benchmark. Even if all JAM instructions were as small as the smallest (pop, one additional SPARC instruction) there would be 52 SPARC instructions in the loop. But most JAM-instructions are more complex than that; the instruction `enterlocal` executes at least 30 SPARC instructions. By using the UltraSPARC instruction counter I have found that JAM in total needs 240 SPARC instructions to execute the inner loop of the benchmark.

**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 it does not then 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 otherwise jump back to label 3.

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.
3.1. PERFORMANCE ON A SEQUENTIAL BENCHMARK

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

```beamer
len_2:
39: func_info(length,len,2)
  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 measurements show that it only takes 47 SPARC instructions to execute the inner loop for BEAM.

*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 I show you how process suspension is handled (Code 6) and finally in Code 7 how bad arguments are handled.

Code 4 (SPARC code for the loop in len/2)

```sparc
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
```
\section*{CHAPTER 3. MEASUREMENTS}

```
.length_len_2_3: ! No suspension
  srl %r3, 28, %r1 ! 5
  subc %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
  ldw [%r4+4], %r4 ! 13
  or %r2, 1, %r2 ! 14
  subc %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
  addc %r1, 16, %r2 ! 19
  bvz, 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. HiPE does 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 explicitly. The native code has 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 the execution keeps on looping. For HiPE generated native code the inner loop is just 27 SPARC instructions.
3.1. PERFORMANCE ON A SEQUENTIAL BENCHMARK

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 the code checks 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)
    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:
    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
```

3.1.2 Speedup

If for each system $S$ we 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: $T_{JAM} \approx 63.32 \times 10^6$, $T_{BEAM} \approx 14.07 \times 10^6$, $T_{HIPE} \approx 3.93 \times 10^6$. 
3.1. PERFORMANCE ON A SEQUENTIAL BENCHMARK

![Figure 3.1: Performance ratio for the benchmark Length.](image)

This gives us the performance ratios $T_{JAM} = 1$, $T_{BEAM} \approx 4.5$, $T_{HIPE} \approx 16.1$ as illustrated in Figure 3.1.

3.1.3 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 3.2), 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 I am 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 [26].
Figure 3.2: The number of executed instructions (in millions) and execution time (in millions of clock cycles) for the benchmark Length.

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.

3.1.4 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 3.3: Total execution time and pipeline stalls for the benchmark *Length*

Figure 3.3 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 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.

3.1.5 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. I can show this with four simple 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.
3.1. PERFORMANCE ON A SEQUENTIAL BENCHMARK

![Graph showing speedup ratio of execution times for benchmarks with different types of calls in HiPE compared to JAM.]

Figure 3.4: Speedup ratio of execution times for benchmarks with different types of calls in HiPE compared to JAM.

\[
\text{% Here we do a tail call to another function.} \\
\text{\texttt{tail_call_other1(X)}} \text{ when } X > 0 \rightarrow \\
\text{\texttt{tail_call_other2(X-1);}} \\
\text{\texttt{tail_call_other1(0)}} \rightarrow \\
\text{0.}
\]

\[
\text{\texttt{tail_call_other2(X)}} \text{ when } X > 0 \rightarrow \\
\text{\texttt{tail_call_other1(X-1);}} \\
\text{\texttt{tail_call_other2(0)}} \rightarrow \\
\text{0.}
\]

\[
\text{% Here we do an ordinary call to another function.} \\
\text{\texttt{non_tail_call_other1(X)}} \text{ when } X > 0 \rightarrow \\
\text{\texttt{non_tail_call_other2(X-1)+1;}} \\
\text{\texttt{non_tail_call_other1(0)}} \rightarrow \\
\text{0.}
\]

\[
\text{\texttt{non_tail_call_other2(X)}} \text{ when } X > 0 \rightarrow \\
\text{\texttt{non_tail_call_other1(X-1)+1;}} \\
\text{\texttt{non_tail_call_other2(0)}} \rightarrow \\
\]

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 I run these benchmarks so that 100,000 calls are made and calculate the speedup ratio between HiPE and JAM as I did for length it is even more evident that tail-recursive calls is what HiPE does best (Figure 3.4). 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 results 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.

3.1.6 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.
3.2. **THE EDDIE BENCHMARK**

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 for HiPE over JAM of more than 8.6 times.

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 mis-prediction 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 tight loop which is HiPE’s greatest strength.

3.2 **THE EDDIE BENCHMARK**

Eddie [8, 28] 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.

My 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.

### 3.2.1 Instructions and clock cycles

<table>
<thead>
<tr>
<th></th>
<th>Cycles (M)</th>
<th>SPARC Instr. (M)</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAM</td>
<td>17.64</td>
<td>12.66</td>
<td>1.39</td>
</tr>
<tr>
<td>BEAM</td>
<td>6.70</td>
<td>4.35</td>
<td>1.54</td>
</tr>
<tr>
<td>HiPE</td>
<td>2.79</td>
<td>2.61</td>
<td>1.07</td>
</tr>
</tbody>
</table>

Table 3.1: 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.

### 3.2.2 Pipeline stalls

Figure 3.5 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.

![Graph](image)

**Figure 3.5:** 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 HiPE is 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.
3.2.3 Different types of calls

<table>
<thead>
<tr>
<th>Call type</th>
<th>Calls</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote tail call</td>
<td>188</td>
<td>0.6%</td>
</tr>
<tr>
<td>Remote call</td>
<td>1,218</td>
<td>3.7%</td>
</tr>
<tr>
<td>Local call</td>
<td>4,125</td>
<td>12.6%</td>
</tr>
<tr>
<td>Local tail call</td>
<td>25,231</td>
<td>77.3%</td>
</tr>
<tr>
<td>BIF call</td>
<td>982</td>
<td>3.0%</td>
</tr>
<tr>
<td>BIF tail call</td>
<td>545</td>
<td>1.7%</td>
</tr>
<tr>
<td>Apply call</td>
<td>6</td>
<td>0.0%</td>
</tr>
<tr>
<td>Apply tail call</td>
<td>333</td>
<td>1.0%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>32,628</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Number of different calls in the HTTP parser.

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 3.2.

3.2.4 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.
3.3. \textit{SCCT, THE AXD 301 BENCHMARK}

The built-in function that takes up most of this time (35 \%) is \texttt{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.

<table>
<thead>
<tr>
<th>JAM</th>
<th>Cycles (M)</th>
<th>Instr. (M)</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not in BIF</td>
<td>16.87</td>
<td>12.15</td>
<td>1.39</td>
</tr>
<tr>
<td>In BIF</td>
<td>0.77</td>
<td>0.51</td>
<td>1.51</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>17.64</strong></td>
<td><strong>12.66</strong></td>
<td><strong>1.39</strong></td>
</tr>
<tr>
<td>% In Bif</td>
<td>4.4%</td>
<td>4.0%</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Time that JAM spends in built-in functions when running the HTTP parser.

3.2.5 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 slightly 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 for the benchmark \textit{length}, HiPE and BEAM executes 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 \textit{length}.

3.3 \textit{SCCT, THE AXD 301 BENCHMARK}

I 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 pro-
program has been designed so that it can be upgraded without stopping the application, and particular care has been taken to ensure high reliability [9].

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.

I 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 I 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 the benchmark. The process that runs the benchmark creates another 2 processes for each iteration. I 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.
3.3. SCCT, THE AXD 301 BENCHMARK

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 compiled ERLANG code that is executed in the benchmark. There is also additional code from the run-time system that is executed. 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 my 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. I 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 the modified JAM system.

3.3.1 Instructions and clock cycles

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

We can see (Figure 3.6.) 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.

### 3.3.2 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.

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 3.4.
3.3. SCCT, THE AXD 301 BENCHMARK

<table>
<thead>
<tr>
<th>Call type</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>remote tail call</td>
<td>13,630</td>
</tr>
<tr>
<td>remote call</td>
<td>21,144</td>
</tr>
<tr>
<td>local call</td>
<td>60,112</td>
</tr>
<tr>
<td>local tail call</td>
<td>36,197</td>
</tr>
<tr>
<td>bif call</td>
<td>107,415</td>
</tr>
<tr>
<td>bif tail call</td>
<td>10,830</td>
</tr>
<tr>
<td>apply call</td>
<td>1,498</td>
</tr>
<tr>
<td>apply tail call</td>
<td>1,800</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>252,826</strong></td>
</tr>
</tbody>
</table>

Table 3.4: Number of different calls in SCCT.

Of these only some of the local tail calls 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.

HiPE has a very rudimentary handling of meta calls: it just lets 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.

3.3.3 The impact of the memory hierarchy

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

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 3.7. As we can see the absolute number of stalls is smaller for HiPE than for JAM.

_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 7 (Dynamic tests that in reality are static.)**

```java
if A > 42
    if B > 42 -> never_happens;
        true -> always_happens
    end;
    true -> never_happens
end
```
3.3. **SCCT, THE AXD 301 BENCHMARK**

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.

**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.

**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 3.8 we can see that no matter which implementation we use, about 40% of the time is spent stalling.
Figure 3.8: Pipeline stalls in percent of total execution time.

### 3.3.4 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 I 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 Chapter 4.
3.3. SCCT, THE AXD 301 BENCHMARK

3.3.5 Built-in functions

The HiPE system uses the same built-in functions (BIFs) as the JAM system.\(^1\) I have made the 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, 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.

![Bar chart showing 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.]

\(^1\)When I talk about BIFs in this paper I refer to the built-in functions that are implemented as C functions in JAM system 4.5.3.
Since all three systems use 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 3.9).

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 spends in ERLANG code is the time we can affect with optimizations in the compiler. For HiPE this 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 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 (134321). 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.

**Pipeline stalls for built-in functions** In Table 3.5 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 exe-
### 3.3. SCCT, THE AXD 301 BENCHMARK

<table>
<thead>
<tr>
<th>BIF</th>
<th>Calls (M)</th>
<th>Inst. (M)</th>
<th>Mc</th>
<th>Load</th>
<th>RAW</th>
<th>Store</th>
<th>Misp</th>
<th>IC</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binary_to_list/1</code></td>
<td>2,502</td>
<td>0.6</td>
<td>1.0</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td><code>db_get/2</code></td>
<td>6,096</td>
<td>10.2</td>
<td>13.7</td>
<td>2.7</td>
<td>0.0</td>
<td>0.0</td>
<td>0.9</td>
<td>0.6</td>
</tr>
<tr>
<td><code>db_get_element/3</code></td>
<td>2,400</td>
<td>1.0</td>
<td>2.0</td>
<td>0.5</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td><code>db_match/2</code></td>
<td>300</td>
<td>0.4</td>
<td>0.8</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td><code>db_put/2</code></td>
<td>1,318</td>
<td>11.4</td>
<td>14.0</td>
<td>2.2</td>
<td>0.0</td>
<td>0.1</td>
<td>1.0</td>
<td>0.7</td>
</tr>
<tr>
<td><code>db_update_counter/3</code></td>
<td>1,012</td>
<td>0.4</td>
<td>0.9</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td><code>element/2</code></td>
<td>80,363</td>
<td>5.1</td>
<td>5.1</td>
<td>0.5</td>
<td>0.0</td>
<td>0.0</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td><code>list_to_binary/1</code></td>
<td>2,402</td>
<td>3.1</td>
<td>5.0</td>
<td>0.8</td>
<td>0.0</td>
<td>0.0</td>
<td>0.3</td>
<td>0.7</td>
</tr>
<tr>
<td><code>setelement/3</code></td>
<td>9,316</td>
<td>2.2</td>
<td>2.5</td>
<td>1.0</td>
<td>0.4</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td><code>spawn_link/3</code></td>
<td>200</td>
<td>0.6</td>
<td>1.4</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td><code>split_binary/2</code></td>
<td>1,902</td>
<td>0.4</td>
<td>0.7</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
</tr>
</tbody>
</table>

**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 3.5: 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.

executed while in the BIF, the number of millions of cycles spent stalling (because of load stalls, read-after-write stalls, store buffer stalls, mis-prediction stalls, and instruction cache stalls, respectively). As can be seen from 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).
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.

3.3.6 Conclusion

The SCCT benchmark is a large benchmark, the size of the executed natively compiled ERLANG code is 64KB, and the total size of the 501 called function is in native code is 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 spend about 40% of their total execution time stalls.

This indicates that even though HiPE runs into problems with the instruction cache because of the size of the program, HiPE does not suffer more from this than JAM and BEAM suffers from other types of stalls. The main reason that HiPE does 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.

### 3.4 RELATED WORK

To the best of my knowledge, there are no previous work where large real-world applications implemented in functional programming languages have been studied with low-level performance measurements. The largest benchmark used in the comparison of implementations of functional languages is usually the compiler for the language.

One reason for this is that performance counters have been available only for the last couple of years and since they are very machine specific they are hard to implement in a generic way that is usable on many different machines. Therefore it seems to be very little work that is closely related to mine.

The use of profiling to detect bottlenecks is not in itself a new technique, and is described for example by Bentley [5]. Also, the use of UltraSPARC performance monitors have been described before [6], but they have not been available for use in ERLANG before.
In recent years with the introduction of Java there has been an increased interest in interpreters. Tia Newhall and Barton P. Miller [23] have developed a, as they call it, "representation model for describing performance data from an interpreted execution". They have concentrated on the Java virtual machine and on methods to see how application programs utilizes the VM. They can study such aspects as the number of method calls, the number of object creates and the total execution time for an application. The HiPE system can monitor corresponding aspects of ERLANG programs such as function calls, number of created processes, number of sent messages, and the total execution time. I have chosen to concentrate on the low level aspects of the behaviors of the virtual machines in this thesis, putting my work quite far from theirs.

The behavior of different interpreters have been studied by Romer et al [27] They have compared four different interpreters, MIPSI, Java, Perl, Tcl and their behaviors on large application programs, and used a simulator to study the low level aspects of the behaviors of these interpreters. The interpreters they have studied differs from each other in many aspects and they all interprete different source languages. They have one program that is implemented in each source language and in C, but their goal has not been to compare these interpreters to each other. My goal on the other hand has been to compare the performance of two different emulators with native code.

The low level performance counters on the UltraSPARC has been used to compare C programs with C++ programs [26] They used large real programs but they did not use the same programs for their C benchmarks as for their C++ benchmarks.

The work that has the closest relation to my work was submitted to OOPSLA’97 but evidently not accepted. In this work [7] James K. Doyle, J. Eliot B. Moss, and Antony L. Hosking have compared three methods for execution of Smalltalk. These three methods Bytecode,
Threaded, and Translated corresponds quite well to the methods used by JAM, BEAM, and HiPE. They have also used a large benchmark to examined the effects of cache misses on the three methods of execution. Instead of using performance counters to measure these costs, as I have done, they have formulated a model for the cost of execution. Then they executed the benchmark one time and five times and measured the total execution time for each benchmark, and concluded that their model fitted well to the measured values. The model allowed them to simulate the execution behavior for different sizes of caches and different costs for instruction- and data cache misses. From these simulations they could conclude that when the cost for an instruction cache miss is high and a "moderate-sized" cache is used then a byte-coded execution model can actually be faster than direct execution. In their experiments they used a SPARCstation 2 which has only one level of cache which is a 64KB unified cache. They have not taken stalls from mispredictions into account and only briefly discusses the effects of a superscalar design.

3.5 CONCLUSION

The execution times in millions of clock cycles and millions of executed instructions for each system and benchmark can be seen in Figure 3.6.

The speedup on Length for HiPE compared to JAM has two reasons: HiPE executes less than 12% of the number of instructions JAM executes, and by utilizing the pipeline better, HiPE can execute 1.87 times as many instructions per cycle as JAM. 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. As shown in Table 3.7.

On the HTTP parser, as on length, HiPE and BEAM executes a lot less instructions than JAM, and HiPE has a lower CPI 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.

<table>
<thead>
<tr>
<th></th>
<th>JAM</th>
<th>BEAM</th>
<th>HiPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>1</td>
<td>4.5</td>
<td>16.1</td>
</tr>
<tr>
<td>HTTP</td>
<td>1</td>
<td>2.6</td>
<td>6.3</td>
</tr>
<tr>
<td>SCCT</td>
<td>1</td>
<td>1.4</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Table 3.7: Relative speedups for BEAM and HiPE as compared to JAM. The speedups are calculated from the total execution times in clock cycles.

For the HTTP parser all three systems spends 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 (see Figure 3.8).

For SCCT 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. All three systems spends about 40% of their total execution time stalling on SCCT.

This indicates that even though HiPE runs into problems with the instruction cache because of the size of the program, HiPE does not suffer more from this than JAM and BEAM suffer from other types of stalls. The main reason that HiPE does 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.

<table>
<thead>
<tr>
<th></th>
<th>JAM</th>
<th></th>
<th>BEAM</th>
<th></th>
<th>HiPE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%L</td>
<td>%M</td>
<td>%IC</td>
<td>T</td>
<td>%L</td>
</tr>
<tr>
<td>Length</td>
<td>12</td>
<td>10</td>
<td>0</td>
<td>22</td>
<td>9</td>
</tr>
<tr>
<td>HTTP</td>
<td>23</td>
<td>10</td>
<td>2</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>SCCT</td>
<td>23</td>
<td>12</td>
<td>6</td>
<td>41</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 3.8: Percentage of the execution time spent in the the most common types of pipeline stalls. The columns %L Shows the percentage load stalls. The columns %M shows the percentage Misprediction stalls. The columns %IC shows the percentage of instruction cache stalls. And the T columns shows the total percentage. All numbers are based on the number of executed machine cycles.

The SCCT benchmark is a large benchmark, the size of the executed native code is 64KB, and the total size of the 50l called function is in native code 239 KB. HiPE is only 1.6 times 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.
Hippo: an Inter-Process Optimizer

In this chapter I will present the design and implementation of an inter-process optimizer. I start by describing what I mean by inter-process optimization (IPO), and motivate its use. Then in Section 4.1 I describe and motivate the profiler that I have used. In Section 4.2 I describe the transformations that I perform on the code in order to merge code from two processes that communicates. The Chapter ends with a short survey of related work in Section 4.3 and a conclusion in Section 4.4.

Programs written in an abstract way can not take full advantage of most compiler optimizations of today, since the abstraction not only hides complexity from the user, but also specifics from the compiler. This is especially true for processes. The problem is that the data flow is hidden in a program using processes.

My thesis is that by giving the compiler the possibility to combine the code of a sender and its intended receiver, it is possible to remove much of the overhead of process communication. I call this type of optimization for inter-process optimization or IPO for short.

The method of inter-process optimization that I have developed merges the code that sends a message with the code that receives the message.
CHAPTER 4. HIPPO: AN INTER-PROCESS OPTIMIZER

Many of the ideas behind the design of this inter-process optimizer have evolved through discussions with Sven-Olof Nyström.

I have implemented a prototype inter-process profiler and optimizer for Erlang which I call Hippo. It uses the HiPE system for compilation. Hippo also uses an instrumented version of the JAM emulator to profile process communication in emulated code. This makes the profiling, described in Section 4.1, easy to implement.

The profile information is used to choose candidates for inter-process optimization. These candidates are made up of pairs of program points, where one program point refers to a send statement, and the other refers to the corresponding receive statement. I discuss how the profiling is performed and what characterizes good candidates in Section 4.1.

When a good candidate pair is found, the functions containing the send and the receive are both compiled to ICode by the HiPE compiler. The code is then translated to a superset of ICode called ICode2. Then some of the code from the receiver is merged with the sender and the resulting code is compiled to native SPARC code.

4.1 PROFILING

Hippo uses profiling to find send-receive pairs that can be optimized. Hippo has to:

1. Find a point in a program where a send is performed.
2. Find out at which receive statement this message is received.
3. Ensure that the mailbox of the receiving process is empty.
4. Ensure that the receiving process is in fact suspended at the receive statement found in step 2, at the time of the send.
4.1. PROFILING

4.1.1 Why not use analysis?

A static analysis that could find sender-receiver pairs (that communicate), would make it possible to do these optimizations without having to run profile executions, and might allow us to exclude runtime tests from the resulting object-code.

The analyzer has to ensure that when a specific send is executed the receiver is suspended at a specific place in the code with an empty mailbox. There are two aspects of ERLANG that makes it difficult to make a static analysis that can give us this information:

1. ERLANG is a very dynamic language. There are no static type information declarations in ERLANG that could help the analyzer. New processes can be created dynamically and references to processes can be passed around as arguments, stored in complex data structures, and even passed as messages themselves. Also, the meta calls\(^1\) frequently used in ERLANG make it difficult to be certain which code is executed by a given process. Finally the code in the system can be replaced by hot-code loading making any analysis across module boundaries dependent on the version of the code. This makes it very hard for an analyzer to see what code the processes referred to by the argument to the send can execute.

2. It is necessary to know the states of processes. The analysis must guarantee that the mailbox is empty and that no other process can send a message to the receiver. Further, the scheduling mechanism in the run-time system also needs to be included in the analysis in order to guarantee that the process is suspended at the right point.

\(^1\)A function can be called by just giving the name of the function and a list of arguments as an argument to the built-in operator apply.
These problems will probably force the analyzer to make a very crude approximation of the set of possible places where a message might be received; thus it would still be necessary to add run-time tests to the resulting code.

### 4.1.2 Description of the profiler

The HiPE ERLANG system can run both emulated code and native code at the same time. Hippo takes advantage of this by first profiling emulated code and then compiling to native code.

I have instrumented the emulator to collect information about the receiver of each `send` instruction. At run-time it is easy to find the receiver of a `send` since the receiving process identifier is given as an argument to the `send` statement.

The collected information has two components: information about the destination (`Dest`), and the number of times the instruction is executed (`Times`). The `Dest` field is initialized to `none`, and the `Times` field to zero. When the send is executed, the `Times` field is increased and the receiving process is checked. If the mailbox of the receiver is empty then the program counter (PC) of the receiver is checked; if the PC is equal to `Dest` or if `Dest` is equal to `none` then `Dest` is set to `PC`. Otherwise `Dest` is set to `unknown`.

![Diagram](attachment:image.png)

**Figure 4.1:** The lattice of possible `Dest` values.

This crude way of profiling the `send` only tells us whether a `send` always (during a particular profiling session) fulfills the prerequisites
for the optimizations or not. A process is considered as a candidate for the transformation that follows, only if it is found to always fulfill the prerequisites of the optimization. In the benchmarks I have considered, this seems to provide satisfactory results.

There are two scenarios for which I fear that this profiler will be inadequate. The first is when code is reused by heavy generalization where the same send might be used in completely different contexts sending messages to different processes. This could possibly be remedied by inline expansion and specialization. The other scenario is where a process has several natural states, implemented by a separate function or a separate function clause for each state. This leads to several receives: one for each state. This would need a more precise profiler and a more complex inter-process optimizer than the one I describe here.

I could extend the profiler with the ability to record more than one destination for each send, and the frequencies of each destination. This way Hippo could possibly get enough information to handle the cases that are problematic today. It would at least give Hippo the ability to optimize the most common case when a send has several destinations.

Another extension would be to profile the natively compiled code, this can be done relatively easy by inserting a call to the function that today profiles the send in the emulator. I could also insert a counter in the inter-process optimized code in order to re-optimize the code if the communication behavior changes.

4.1.3 Results

I have tested the profiler on two larger ERLANG programs: a part of AXD 301 [9], a new ATM switch from Ericsson, and a HTTP parser in Eddie [8]. For these benchmarks, all sends are always to processes with empty mailboxes suspended at the same address for each send;
that means that the profiler gives us exactly one specific destination address for each send.

4.2 THE TRANSFORMATION

Given two ERLANG functions \( f \) and \( g \) that constitute a sender-receiver pair Hippo can merge them by inserting code from \( g \) (the receiving code) into \( f \) (the sending code). I will call the new function created by this merging \( f' \), and refer to the sender (the process executing \( f \)) as \( \alpha \) and the receiver (the process executing \( g \)) as \( \beta \).

The function \( f \) can be divided into the following abstract blocks of code:

1. Head (code preceding the \texttt{send})
2. Message creation
3. \texttt{send} (the one indicated by the profiler)
4. Tail (the rest of the code)

The function \( g \) can be viewed as:

1. Head (code preceding the \texttt{receive})
2. \texttt{receive} (the one indicated by the profiler)
3. Tail (the rest of the code)

The intention of the transformation is to allow process \( \alpha \) to execute code that would otherwise have been executed by \( \beta \). Thus the resulting code for \( \alpha \), function \( f' \), will contain fragments of the code from \( g \) (Figure 4.2).

The merged function \( f' \) is a copy of the function \( f \) with these six additions:
4.2. THE TRANSFORMATION

Figure 4.2: Before the merging, function $f$ is executed by $\alpha$ and function $g$ is executed by $\beta$. After the merging, $f^{'}$ is executed by $\alpha$.

1. Test – A test is inserted before the send in $f^{'}$. This test checks whether $\beta$ is suspended at the right program point (at the receive in $g$) with an empty mailbox. If this test succeeds the execution continues with the optimized code (item 2), otherwise the execution continues with the original code of $f$.

2. Message copying – If the test succeeds the message is copied from the heap of process $\alpha$ to the heap of process $\beta$ using an explicit copy instruction.

3. Restore state – All live $\beta$-temporaries are read from the stack of $\beta$.

4. Code from $g$ – The code from $g$ that is suitable for “external” execution is then executed.

5. Save state – All live $\beta$-temporaries are written back to the stack of $\beta$.

6. $f$-tail – A copy of the tail of $f$ is executed.
Since Hippo rely on a subsequent optimization pass to clean things up, the merging is quite easy and straightforward. This optimization pass, which performs a generalized constant propagation and dead-code elimination, will remove unused paths from $g$.

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

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

### 4.2.1 The merged code

Let us look at some of the details of the transformation.

When Hippo first copy the code from $f$ to $f'$ all temporaries are renamed so that they begin with an “a” denoting that they refer to values on the heap of $\alpha$. When the code from $g$ is inserted a “b” is added to the name indicating that they refer to values on the heap of $\beta$.

*Test*  The test in the function $f'$ checks that $\beta$ is suspended at the right `receive` with an empty mailbox.

*Message copying*  To give the code from $g$ access to the message sent from the process $\alpha$ the message is copied from the heap of $\alpha$ to the heap of $\beta$. A reference to the copied message is put in the temporary used by the `receive` in $g$. 
4.2. THE TRANSFORMATION

*Restore state*  When an ERLANG process is suspended, all live temporaries are saved on the stack, in the same way as for a function call. To get access to these temporaries Hippo “restores” the state of the process \( \beta \) by reading the live \( \beta \)-temporaries from the stack of \( \beta \) into temporaries used by \( \alpha \).

*Code from g*  The code from \( g \) has to be rewritten so that it can be executed “externally”, that is, from within process \( \alpha \). This means that all primitives of the language have to be rewritten for external execution.

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

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

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

1. A call to another function, a meta call (apply) or a return can not be extracted since the control could be passed to code that is not adapted for external execution.
2. Instructions that lead to the suspension of the process, such as the explicit suspension instruction or a receive.
3. Some built in functions are large and uncommon and it is not worth the effort to make special versions that can handle two heaps.

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

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

**Save State** To propagate changes in the state of $\beta$ Hippo has to save the new state at the end of the extracted code. To this end, Hippo writes all live temporaries back to the stack at the end of each path of the extracted code. At the end of each of these paths the continuation pointer of $\beta$ is set to point to a stub, containing the instructions from that path that could not be extracted from $g$.

**$f$-tail** To simplify optimization Hippo duplicates the tail of $f$. From the end of each path of the extracted CFG Hippo inserts a jump to this copy. This ensures that when the code in the copy is reached, the execution is guaranteed to have passed through the code extracted from $g$. 
4.2. THE TRANSFORMATION

4.2.2 Further Considerations

The garbage collector in HiPE requires that all data structures accessed by a process are allocated on the heap of that process. This invariant is temporarily broken while the process $\alpha$ accesses the state of process $\beta$, but since hippo has control over when $\alpha$ is suspended and when garbage collection is triggered, Hippo can ensure that the invariant is maintained at these points.

Another problem is that normally, the heap of a process is only garbage collected when it is running. With IPO data may be allocated on the heap of a suspended process making it run out of heap space. To handle this Hippo can explicitly run the garbage collector on the suspended process if it runs out of heap space.

Since Hippo will change the scheduling behavior, one might suspect that this could lead to a change in the concurrency semantics of the program. Since Hippo does not allow the code from $g$ to loop, and does not count any reductions other than those counted without the optimization, the observable behavior will remain unchanged.

ERLANG supports hot-code loading where a module can be replaced at run-time. The inter-process optimizer will merge code from two functions ($f$ and $g$). If the module of $g$ is updated then old code from $g$ is kept inside $f$ (actually in $f'$). Fortunately there is no risk that the old code might be executed instead of the new code. This is ensured by the run-time test in $f'$ which will only succeed when the receiver is suspended from old code. (If the module containing $f$ is replaced then all optimized code is removed and there is no problem at all.)

4.2.3 Return messages

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

1. There is a send in the function \( g \)-tail.
2. The destination of the send in \( g \) is the process \( \alpha \).
3. All paths through \( f \)-tail contains receive.
4. The mailbox of \( \alpha \) is empty.

Hippo ensures this by always checking that the mailbox of \( \alpha \) is empty before the optimized code is used. By doing this check at the beginning Hippo gets a very simplified CFG for \( f' \).

Then Hippo just copies the message from the heap of process \( \beta \) to the heap of process \( \alpha \) if the destination of the send is \( \alpha \). Hippo does this by the following rewriting of the merged code from \( g \):

**Example 8 (Transformation of a send in \( \beta \).)**

\[
\begin{align*}
\text{:=} & \ \text{op\_send}(bv1, bv2) \\
\implies & \\
bv3 & := \text{op\_self()} \\
& \text{if} \ \text{if} \ \text{op\_exact\_eqeq\_2}(bv1, bv3) \ \text{then} \ 1 \ (0.99) \ \text{else} \ 2 \\
1: & \ av1 := \text{external\_copy}(bv2) \\
& \text{av2 := } \text{\_true/} \\
& \text{goto 3} \\
2: & \ := \text{op\_send}(bv1, bv2) \\
3: & 
\end{align*}
\]
4.2. THE TRANSFORMATION

The first receive in each path of the tail of \( f \) is then rewritten as:

**Example 9 (Transformation of a receive in \( \alpha \))**

\[
\text{av3} := \text{get_msg}()
\]

\[
\Rightarrow
\]

\[
\text{if op\_exact\_eqeq\_2(\text{av2}, /true/) then 4 (0.99) else 5}
\]

\[
4: \text{av3} := \text{av1}
\text{goto 6}
\]

\[
5: \text{av3} := \text{get\_msg}()
\text{goto 6}
\]

Here \( \text{av2} \) is used as a flag to indicate that the message has been passed directly with a copy.

Now, the nice thing is that by using *generalized constant propagation* Hippo can often remove the tests completely and end up with code like:

**Example 10 (Optimized code for a return message.)**

\[
1: \text{av1} := \text{external\_copy}(\text{bv2})
\text{goto 3}
\]

\[
3: 
\]

\[
4: \text{av3} := \text{av1}
\text{goto 6}
\]

and depending on how \( \text{av3} \) is used Hippo might get rid of the copying between the processes completely.

4.2.4 An example

Let us look at an example of the inter-process optimization. Assume we have the functions \( f \) and \( g \) as shown below, where \( f \) is executed by the sending process \( \alpha \) and \( g \) by the receiver \( \beta \). Here the process
\( \beta \) has a state given by the variable \( S \) in \( g \). A message to \( \beta \) of the integer 1 or the integer 2 forces the process to move to the corresponding state. Any other messages makes the function \( g \) return the information about the state to the caller\(^2\) (not to the sender though).

\[
f(P) \rightarrow \\
P \mid 1, \\
ok.
\]

\[
g(S) \rightarrow \\
\text{receive} \\
1 \rightarrow \\
g(1); \\
2 \rightarrow \\
g(2); \\
_ \rightarrow S \\
\text{end.}
\]

In ICode these two simple functions become somewhat more complex. In \( g \) the receive is broken up into a test whether the mailbox is empty or not, an explicit suspension, and an instruction to retrieve a message from the mailbox (See Example 12). The pattern matching in the receive is also made explicit by the two tests whether the message is the integer 1 or 2.

**Example 11 (ICode for \( f \).)**

\[
f(v1) \rightarrow \\
1: \ v3 := /1/ \\
\quad v4 := \text{op\_send\_2}(v1, v3) \\
\quad v5 := /ok/ \\
\quad \text{return}(v5)
\]

\(^2\)If \( g \) is the top-level function of \( \beta \), then \( \beta \) will terminate quietly, losing \( S \)
Example 12 (ICode for g.)

\[
g(v1) ->
5: \quad := \text{redtest}_0()
goto 4
4: \quad \text{if mbox_empty()} \text{ then } 6 (0.50) \text{ else } 7
7: \quad v5 := \text{get}_mmsg()
goto 8
8: \quad \text{if is_integer}, 1(v5) \text{ then } 9 (0.50) \text{ else } 1
1: \quad \text{if is_integer}, 2(v5) \text{ then } 10 (0.50) \text{ else } 2
2: \quad := \text{join()}
\quad \text{return}(v1)
10: \quad := \text{join()}
\quad v1 := /2/
goto 5
9: \quad := \text{join()}
\quad v1 := /1/
goto 5
6: \quad := \text{suspend}_msg()
goto 7
\]

If we just merge these two functions without doing any further optimizations, we end up with a lot of code; see Example 13.

First we have to add a test to see whether the argument \((av1)\) really is a process identifier \((\beta)\). This test is implicit in the send operator but we need to be sure in order to get the process control block of the other process. If the argument is not a PID then we go ahead and do the send anyway and let the send deal with that problem.

In ICode2 we have not only an implicit stack and heap for \(\alpha\) but also for \(\beta\). These are fetched with the get pcb instruction. Then we can check the state of \(\beta\) against a magic number that is unique for each receive suspension point. In this case the number is 3.

We also check if the mailbox of \(\alpha\) is empty, this is actually unnecessary here since we do not expect a return message but for simplicity in my prototype I always add the test.

If everything works out we read the internal state \((bv48, 8)\) of \(\beta\). Then we switch on the message \((L17, L19)\), if it is 1 or 2 the internal
state is saved (L25). The suspension point state of $\beta$ is also set to the magic number 3, before the execution continues back in the code from $f$ (L14).

If the message was something but 1 or 2 (L24) then the internal state of $\beta$ is saved and the continuation pointer of $\beta$ is set to L26. When process $\beta$ is scheduled next time it will then jump to L26 restore it’s state and return it (L27).

The execution of $\alpha$ continues at L14 where the atom $\text{ok}$ is returned (L14).

Example 13 (The merged function $f'$.)

\[
f'(\text{av1}) \rightarrow\]

1: \text{av3 := /1/}\n   \text{if is_pid(\text{av1}) then 32 (0.99) else 35}\n35: \text{av4 := op_send_2(\text{av1}, av3)}\n   \text{goto 29}\n29: \text{av5 := /ok/}\n   \text{return(\text{av5})}\n32: \text{av47 := av1}\n   \text{:= get_pcb(\text{av47})}\n   \text{av50 := external_get_state(\text{av1})}\n   \text{av51 := /3/}\n   \text{if op_exact_eqeq_2(\text{av50}, \text{av51}) then 33 (0.99) else 35}\n33: \text{if mbox_empty() then 34 (0.99) else 35}\n34: \text{av43 := av3}\n   \text{av4 := av3}\n   \text{goto 15}\n15: \text{bv48 := external_restore()}\n   \text{goto 16}\n16: \% 'Copy msg A -> B'\n   \text{bv49 := external_copy_1(\text{av43}, \text{av47})}\n   \text{goto 17}\n17: \text{if is_integer,1(\text{bv49}) then 18 (0.50) else 19}\n19: \text{if is_integer,2(\text{bv49}) then 21 (0.50) else 22}\n22: \text{goto 24}\n24: \% 'Save Bs state, set BCP'\n   \text{external_save([\text{bv48}])}\n   \text{external_set_cp(\text{avvar,47, L26})}\n   \text{:= external_set_state(\text{/0/})}\n   \% 'Return to As code'\n   \text{goto 14}
4.2. THE TRANSFORMATION

14: av5 := /ok/
   return(av5)
21: bv48 := /2/
   goto 20
20: := redinc_0()
   goto 23
23: goto 25
25: % 'susP point, save a consistent state'
   % state,3,const,3
   := external_set_state(/3/)
   external_save([bv48])
   % 'susP_msg'
   goto 14
18: bv48 := /1/
   goto 20
26: % 'B_contPoint restore consistent state'
   bv48 := restore()
   goto 27
27: return(bv48)

This code can then be optimized with generalized constant propagation. In the optimized code (see Example 14) there is only one path through the code of $g$ left. The tests whether we can do the optimization are still there but the copying of the message and the switching as well as unused paths are removed.

Example 14 (Optimized merged code)

$f'(av1) \rightarrow$
1: av3 := /1/
   if is_pid(av1) then 32 (0.99) else 35
35: av4 := op_send_2(av1, av3)
   av5 := /ok/
   return(av5)
32: av47 := av1
   := get_pcb(av47)
   av50 := external_get_state(av1)
   av51 := /3/
   if op Exact_eqeq_2(av50, av51) then 33 (0.99) else 35
33: if mbox_empty() then 34 (0.99) else 35
34: bv48 := external_restore()
   bv48 := /1/
   := redinc_0()
   := external_set_state(/3/)
external_save([bv48])

% 'SuSp_msg'

av5 := /ok/

return(av5)

26: % 'B_contPoint restore consistent state'

bv48 := restore()

return(bv48)

4.3 RELATED WORK

The idea of inter-process optimization in itself is not new, probably as old as the concept of concurrent programming. In 1990 for example McNamee and Olsson [21] described and evaluated a number of source-level transformations for optimizing process communication in imperative languages. They did not, however, describe how the optimizations could be integrated in a compiler.

In Concurrent logic programming (CLP) and concurrent object-oriented programming (COOP) programs tend to have many small processes and intensive process communication. There are situation where one process need not be started until another process has terminated, if this situation can be detected in advance then the cost of concurrency can be reduced. Work have been done to detect this with analysis [19, 20].

The problem of finding when and where to perform inter-process optimization is similar to that of when and where to optimize dynamic dispatch in the object-oriented programming language Self. According to Agesen and Hölzle [1] the precision of profiling is similar to that of static analysis (concrete type inference) for the Self case. Still, it is not certain that this conclusion will hold for process communication in ERLANG.

Plevyak, Zhang and Chien [25] describe a technique for inter-process optimization that is somewhat similar to mine. Their optimization of communication between (concurrent) objects in a concurrent object-oriented language uses a run-time test to determine whether the op-
timization can be applied, just like mine. And they inline method operations from one object into another, much like I merge code from the receiver into the sender. On the other hand, they only need to handle static messages since the name of each message is provided in the language they are using, while I handle dynamically created messages. They also have to use static type analysis to get the code of the receiving object, while I use profiling and my technique can handle processes that may execute code that is not present when the optimization is applied.

4.4 CONCLUSION

4.4.1 Potential gains

Inter-process optimization can reduce the overhead of process communication in four different ways.

4.4.2 Reduced message passing

It is common in Erlang programs that a process creates a message, sends it (by copying it) to another process, which subsequently performs some matching on the form of the message, accesses some components of the message and never looks at the whole message again.

The creation and copying of the message to be sent can be avoided by short-circuiting the switching on the received message. This can also save time in the garbage collector.

4.4.3 Short-circuited switches on messages

IPO can create a control flow graph with less branches and fewer instructions to execute by using the information about the form of the message to short-circuit the pattern matching in the receive.

It will also make the hardware prefetching mechanisms work better. If several different messages with the same frequency are received by
the same receiver, then the switch will go in different ways each time rendering the prediction useless, which results in pipeline stalls.

### 4.4.4 Reduced context switching

IPO can, in the cases where the receiver immediately answers, remove the context switch completely. This not only means that the receiver does not need to be scheduled, but it also means that the executing process does not need to be suspended. Measurements indicate that in most concurrent ERLANG programs the processes do not use their whole time-slice instead they are suspended on \texttt{receive}. If the sender can keep on running until the time-slice is used up then the expensive scheduler would have to be executed less. To let the same process execute longer gives better cache behavior.

### 4.4.5 Enabling of further optimizations

The real gain of IPO comes from the ability to do optimizations on the merged code, just as the real gain from procedure inlining comes from the optimizations done after the inlining. We get the possibility to do constant propagation, common subexpression and so on, on code both from the sender and the receiver. IPO will also make it possible to do register allocation across process boundaries, something that has not been possible before.

### 4.4.6 Results

I have used Hippo on small benchmarks with simple communications and been able to see that the messages do arrive, and in the right order even when several processes are involved. I have also tested hot-code loading and concluded that it also works.

Since the implementation is not finished yet, I have not been able to test this optimization on any larger programs. I can therefore not
say how big impact this optimization might have on typical ERLANG programs.

The overhead for message passing and context switching is very small in ERLANG so the gain from just merging the code from the receiver with the code from the sender will be small. To execute the loop and send 1,000,000 messages from \( \alpha \) to \( \beta \) and back with the benchmark in Example 15 takes, on a 140MHz UltraSPARC I running Solaris (Sun OS 5.6), on the average 21.6 seconds for JAM, and 14.3 seconds for HiPE. This indicates that it takes less than 7 micro seconds to send these messages in native code. Hippo can not at the moment handle return messages properly, and can therefore not really compete on this benchmark.

Example 15 (A small benchmark)

```erlang
loop(Pid, 0) ->
    Pid ! 0, self(),
    0;
loop(Pid, N) ->
    f(Pid),
    loop(Pid, N-1).

f(Pid) ->
    Pid ! 1, self(),
    receive
        reply -> ok
    end,
    42.

g(S) ->
    receive
        0,P2 -> S;
        1,P2 -> P2 ! reply,
    g(1);
    -> what
end.
```
From the tiny benchmarks that Hippo can execute, I have observed everything from significant slowdown to speedups of several times compared to native code compiled with HiPE.

The slowdown comes from an artificial benchmark that just sends 1,000,000 simple messages to a receiver that does not reply. Without IPO this will be handled in batches of about 4,000 messages (the size of the time-slice). With IPO each message is sent and received at a time, giving 1,000,000 cheap accesses to the sender instead of 500 expensive context switches. I do not believe that it is typical for ERLANG programs to just send thousands of messages without waiting for a reply.

Another reason for the slowdown seems to be that the prototype cannot use a stack trimming optimization usually used in HiPE since it makes it harder to keep a consistent state between the stubs of $g$. In the final version I hope to have corrected this.

The third reason for the slowdown is that the prototype does yet not remove the unnecessary creation of the message.

When I have fixed these problems and completed my prototype, I hope to be able to get the overhead down to a level where a remote procedure call to another process can be almost as cheap as a function call.
Chapter 5

Conclusion

5.1 Measurements

In this thesis I have compared three ERLANG run time systems; JAM, BEAM and HiPE. They all have different execution models: The JAM system uses a stack-based byte-code interpreter. The BEAM system uses a directly threaded implementation of a virtual register machine. The HiPE system uses direct execution of native SPARC code. But they share large parts of the run-time system such as the garbage collector and the built-in functions. HiPE uses the same front end as JAM, while BEAM uses a more sophisticated front end with for example better compilation of pattern matching.

To compare these systems I have instrumented the emulators and the run-time systems in order to be able to use low-level performance counter available on UltraSPARC. I have then made the measurements on three benchmarks: Length, The Eddie HTTP Parser, and SCCT.

The Length benchmark is a small sequential benchmark that only consists of two nested loops that traverses a list. On this benchmark HiPE is over 16 times faster than JAM and over 3 times faster than BEAM. The overhead for interpretation is evident in this benchmark where HiPE only needs 6 million instructions and JAM and BEAM need 49 million instructions and 10 million instructions respectively.
JAM and BEAM also have problems (10 % and 14 % of their execution time) with pipeline stalls from mispredictions.\(^1\) All three systems have some problems with load stalls (a value is needed before it has been completely loaded into a register) but none of the systems have troubles with instruction cache misses.

The Eddie HTTP Parser is a mildly concurrent benchmark which parses HTTP requests, there are only five message sends during one execution of the benchmark. The parser uses some built-in functions (4.7% of the calls are calls to built-in functions). On this benchmark HiPE is over 6 times faster than JAM and over 2 times faster than BEAM. Here BEAM’s better front end almost makes up for the overhead of emulation: BEAM executes 4 million instructions which is not much more than HiPE’s 3 million executed instructions and a lot less than JAM’s 13 million executed instructions. All three systems have about the same percentage of pipeline stalls: 36% for JAM and HiPE, and 37% for BEAM.

The SCCT benchmark is a time critical part of Ericsson’s new ATM switch, AXD 301. This benchmark is slightly more concurrent than the HTTP Parser, with several processes and over 3000 message sends. It also uses the built-in functions heavily about 32% of the execution time for JAM is spent in built-in functions. SCCT is a large benchmark, the size of the native code that is executed by the benchmark (not counting code in the run-time system) is about 64KB. On this benchmark HiPE is only about 1.6 times faster than JAM and only about 1.1 times faster than BEAM. Here the better front-end of BEAM completely makes up for the overhead of emulation, indicated by the fact that BEAM and HiPE both execute about 62 million instructions on this benchmark.

\(^1\)A pipeline stall is when the execution pipeline in the processor is not moving forward, for example because the direction of a branch has been mispredicted.
Native code outperforms emulated code on small tight loops of sequential code, even compared to code with a better front end. On larger benchmarks that uses concurrency and built-in functions the advantage is smaller. In order to make a high performance ERLANG implementation that is of real benefit to the telecom community one has to consider these aspects and come up with solutions that can ensure large speedups even for large concurrent programs.

The goals with the measurements were to:

- Examine the performance of the code generated by the HiPE compiler in order to identify possible bottlenecks.
- Compare emulated code with native code when executing very large programs, and to answer the question whether native code would suffer from instruction cache misses.

The measurements have shown us that the speedup for HiPE over JAM varies significantly depending on the program, from 16 times for the length benchmark to 1.6 times for the SCCT benchmark. I believe that HiPE faces three bottlenecks:

1. Built-in functions. In SCCT a large portion of the time is spent in built-in functions, this time will not be decreased by low-level optimizations on the ERLANG code.

2. The scope of optimizations. In HiPE one function is compiled at a time and most optimizations are done on a basic-block or extended basic-block level. This makes the scope of the optimizations very small. In the length benchmark, which only consists of two looping function, the compiler had access to half the program at a time (one function). In SCCT (with 501 executed functions) the HiPE compiler only has access to one 501th of the program at the time. It would probably not be
feasible to optimize the whole program at a time, but for example by utilizing inlining the scope of the optimizations could be increased.

3. Only low-level optimizations. Today HiPE only performs low-level optimizations and for example pattern-matching is not as efficient as in BEAM.

To be able to handle really large programs containing large pieces of code that is not time critical, for example error handling and maintenance code, I believe that a combination of native and emulated code is needed. HiPE already supports this today but since JAM and HiPE use different stack layouts the costs for switching between the execution models are unnecessarily high.

In order to answer the question whether native code would suffer from instruction cache misses I have compared the native code with emulated code for a large program and can conclude that:

- Yes – HiPE does stall because of instruction cache misses.
- No – it is not a disadvantage compared to emulated code which suffers from load stalls and misprediction stalls instead.

5.2 INTER-PROCESS OPTIMIZATION

My goal with the inter-process optimizer and profiler was to examine whether ERLANG programs show a communication behavior that is consistent enough to allow inter-process optimizations and in that case present a method for performing inter-process optimization.

I have designed and implemented an ERLANG process communication profiler that is simple and fast (it has a low overhead). The results from profiling the benchmarks presented earlier are encouraging: all message sends behave in a consistent way that enables the optimization.
5.2. *INTER-PROCESS OPTIMIZATION*

My optimization method merges code from the sending process with code from the receiving process. This method can be used in conjunction with the profiler to implement inter-process optimization in a safe way even in an environment with hot code loading.

I have implemented a prototype inter-process profiler and optimizer, Hippo. My experiments with Hippo have shown that the method works even though it is not yet certain how much the execution time can be reduced.
apply /3, 13
atom, 7

bignums, 7
binary, 7
branch following, 33

cache
   address, 30
   index, 30
   offset, 30
   tag, 30

cache access stage, 32
catch, 13
committed, 32
cons, 8
CPI, 29
cycles per instruction, 28
data cache, 30
decode stage, 32
dispatch stage, 32

Erlang, 6
Erlang Term Storage, 14

ETS, 14

ETS-tables, 14
execution stage, 32
export, 8
external cache, 29
fail point, 23
failpoint, 40
fetch stage, 32
fixnums, 7
floats, 7
FPU, 31
gcc, 19
GET request, 53
grouping stage, 32
GRU, 31
handle, 13
heap, 15
hot-code loading, 9
IEU, 31
index, 30
instruction cache, 30
instruction fetch, 32
instruction level parallelism, 28
INDEX

instructions per cycle, 28
Integer pipe wait stage, 32
inter-process optimizer, 75
IPC, 28
IPO, 75
likely taken, 33
lists, 8
load miss stage, 32
mailbox, 11
major time-slice, 17
message, 10
message passing, 10
meta call, 13
module, 8
next field, 33
nil, 8
node, 13
not likely taken, 33
not taken, 33
OTP, 14
pattern matching, 8
PCR, 34
PIC, 34
PID, 7
Pipelining, 31
ports, 7
process, 10
process control block, 15
process table, 16
RAW, 35
ready queue, 16
record, 7
reductions, 17
references, 7
remote call, 9
scheduler, 16
stack, 15
suspended, 11
tail-recursion, 7
taken, 33
throw, 13
time-slice, 17
timeout, 11
trap resolution stage, 32
tuples, 8
writeback stage, 32
REFERENCES


REFERENCES


REFERENCES


Real Time Applications. In 8th SETSS, *Florence, Italy, March 30 – April 1, 1990*. 
Computing Science Department
Information Technology
Uppsala University
Box 311, SE-751 05 Uppsala, Sweden

Uppsala Theses in Computer Science 32
ISSN 0283–359X