Erlang on Adapteva's Parallella

Magnus Lång
Abstract

Erlang on Adapteva's Parallella

Magnus Lång

By connecting many simple general-purpose RISC CPUs with a Network-on-Chip memory system, the Epiphany co-processor architecture provides promising power-efficiency. This thesis presents ParallErlang, a modified Erlang Runtime System, capable of running some actors on the Epiphany co-processor. The complete lack of caches is dealt with by introducing an Epiphany backend to the HiPE Erlang compiler, and a software implementation of an instruction cache. Memory system inconsistency is dealt with by constructing a sequence of instructions with fence semantics, and having HiPE inline this “fence” where required. Finally, performance and power-efficiency is measured and evaluated, and while no benchmark show any improvement over an ARM Cortex-A9 CPU, benchmarks also indicate that should overheads be possible to eliminate, an improvement of over two orders of magnitude could be possible, bringing power-efficiency superior to the ARM.
1 Background

Network-on-Chip (NoC) architectures\cite{19} use asynchronous networks rather than conventional buses to implement their memory systems. Memory requests and responses are routed through the network to their destinations. They promise improved power efficiency and scalability compared to the conventional memory designs\cite{18}. However, the lack of any central authority often implies they give weaker ordering guarantees.

Adapteva’s Epiphany is a general-purpose manycore architecture designed to push the performance per watt envelope by utilising many simple cores connected by a NoC. At the time of its reveal, the 64-core Epiphany-IV bested the most efficient GP-GPU on the market, Nvidia’s Kepler, in FLOPS/W. The architecture is intended to be used as a coprocessor, like a GPU. But, unlike a GPU, it is programmed using typical C code. However, programming it is not exactly like programming a traditional CPU. The lack of a cache forces the programmer to manually manage the locality of data in the small SRAM that lies alongside each core.

Since this architecture provides such promising power efficiency, it is naturally interesting to investigate the ability to use higher-level programming languages. Actor programming languages, such as Erlang, where the units of concurrency, called processes in the Erlang case, do not share data, seem like a good fit for this architecture since the data locality problem is much simplified because of that. This thesis presents an adaptation of the Erlang Runtime System (ERTS) that runs on the Parallella development platform, and allows running Erlang processes on the Epiphany. The next chapters will describe how the challenges of the Epiphany platform were overcome and presents performance and power-efficiency evaluation and comparison.

1.1 The Parallella

Adapteva’s Parallella (figure 1) is the Epiphany development platform used for this project. It is a credit-card-sized computer on a chip, and uses a Zynq-7000 series ARM SoC, which contains a dual-core ARM Cortex-A9 CPU running at 667MHz\cite{7}, and configured with 32KB L1 data and instruction caches and a 512KB L2 cache\cite{29}, as well as some FPGA programmable logic. The board also sports the 16-core Epiphany-III co-processor, running at 600MHz. Although a model with the 64-core Epiphany-IV was designed, it never entered mass production. The ARM CPU has a maximum power consumption of 3W, while the Epiphany has a maximum power consumption of 2W \cite[sec. 8.4]{7}.

The Epiphany architecture (figure 2) is notable not only due to its high power efficiency, but also due to its use of a Network on Chip (NoC) memory system. Instead of a single, or a hierarchy of, memory buses, Epiphany routs memory transactions as packets through a grid-shaped network. The 32-bit addressable memory space is divided into a 64 by 64 grid of 1MB sections, and each physical core occupies one such section.

As the figure shows, there are actually three networks implementing the memory system on the Epiphany. The first one, called the \texttt{cmesh}, carries write transactions and replies to read transactions, but only those that are bound between two cores on the same die, and is clocked the fastest of the three, at one packet per CPU cycle. Since a packet can contain up to 64 bits of data,
Figure 1: Adapteva’s Parallella

Figure 2: The Epiphany Architecture
this yields a theoretical goodput of 38.4 Gbps per link \[4, \text{ sec. 5.1}\]. The other two networks are the \textit{rmesh}, that carries load transactions, and the \textit{xmesh}, that carries write transactions and replies to read transactions that are bound for or coming from outside the chip. These two networks are clocked at an eighth of the rate of the \textit{cmesh}.

The Parallella is also equipped with a 1GB RAM chip. We will refer to this memory as DRAM, as that is the technology used to implement it. This will be contrasted to the local memory beside each Epiphany core, which is of the type, and will be referred to as, SRAM. \textit{rmesh} and \textit{xmesh} leave the Epiphany chip through buses called e-link, one in each cardinal direction. On the Parallella, the east e-link is connected to the FPGA in the Zync, which is typically programmed to provide a memory controller mapping a section of the Epiphany's address space to DRAM. This section is 32MB by default, and is referred to as the \textit{shared memory} region. The FPGA is also programmed to provide a mapping of Epiphany local memory and memory-mapped registers into the ARMs physical address space, which is how the Epiphany can be programmed and communicated with.

1.2 Erlang and the Erlang Runtime System

Erlang is a functional, concurrent programming language. It is used to program telephone switches and server software for banking systems, instant messaging systems, and more. Erlang is designed for building scalable, fault-tolerant systems, and that purpose has guided much of the design of the language. In Erlang, the units of concurrency do not share any state. For this reason, they are called \textit{processes}. As they do not share any state, one process crashing does not affect another. However, if two Erlang processes are dependent in such a manner that both of them should be restarted if one of them crashes, they can be \textit{linked} together. When a process crashes, all other processes linked to it will be sent \textit{exit signals}, causing them to also crash. This is desirable to make crash recovery clean, leaving no process in a useless state. In order to communicate, Erlang processes use \textit{message passing}. A message can be any value, called \textit{terms}, and messages are added to the end of a receiving process’ mailbox. Pattern matching is a central concept in Erlang which is found in many functional languages. Patterns can not only be used to deconstruct values and to select one of several code paths, for example in case-statements, but can also be used to \textit{receive} messages from the process’ mailbox out of order, if the first messages in the mailbox matches none of the patterns in a receive statement.

Erlang is a managed language, it is compiled to a byte-code which runs on the BEAM virtual machine, and its memory is managed and garbage collected by the Erlang Runtime System. The bytecode is architecture-agnostic, and a BEAM-file will run on any architecture the Erlang Runtime System has been ported to, without recompiling. As Erlang processes do not share any data, garbage collection is also isolated to a single process, and does not need to disturb other processes. In order to support lightweight concurrency, the Erlang Runtime System performs its own scheduling. Instead of creating a new thread every time an Erlang process is started, there is a fixed number of \textit{scheduler} threads, that runs an Erlang process until it blocks or has exhausted its time-slice, and then picks another waiting Erlang process to run.

Erlang terms are constructed from a fixed number of types. These include
numbers, lists, tuples, process identifiers, binaries (binary data), and atoms. The atoms are a type that can also be found in Prolog and Lisp, although they are called “symbols” in Lisp. Atoms are a kind of strings, but with the additional properties that they can be very efficiently compared for equality and that they are space efficient. They achieve this by being “interned,” the text inside atoms are stored in a table shared by all processes in the Runtime System, and all atoms that are the same also share a single entry in the table. Thus, comparing their entry numbers from this table is sufficient to compare equality between two atoms.

In listing 1, an Erlang module, stats, provides an example of a concurrent Erlang program. A process started by stats:start_link/0 keeps track of the numbers sent to it with stats:add_number/2, allowing their current average to be queried at any time with stats:get_average/1. This could be used in a web server to count the average time taken to service a request, or in a telephone exchange to count the average duration of a call. Note how variables are written with an uppercase letter, atom literals with a lowercase letter, and tuples are written using curly braces. start_link starts a new process by passing an anonymous function to spawn_link, a builtin function that creates a new Erlang process and links it to the current one. add_number sends the number in an Erlang message using the ! operator to a process identifier previously created by start_link. get_average additionally includes its own process identifier, from the self() function, so that the server knows where to send the response,
and uses a receive statement to filter out messages starting with the atom `avg`, so that other messages sent to the process that is calling `get_average` are not accidentally received instead. The server loop is the tail-recursive `loop` function. Using a receive statement, the two different messages from the two functions above are expected, and are handled differently. Each clause of the receive statement ends with a tail-recursive call to itself. That way, the server does not stop, by returning from its main function, after a single message.

The atom table is not the only shared resource in the Erlang Runtime System. There is also the module table, of all loaded modules, containing data literals, the bytecode of the module, etc. There is the process table, which map process identifiers to the Process Control Blocks (PCBs). The PCBs in turn keep all the state of a process, its stack and heap, instruction pointer, and so on. Binaries, the data type containing raw binary data, can also be shared between processes if they are large enough. In this case, binaries are stored separately, with a reference counter of how many processes have a reference to it. This speeds up Erlang programs that process large amounts of data, as sending it from one Erlang process to the next does not require copying it. There is also a process registry. This feature is a global table mapping names, in atom form, to process identifiers.

1.3 Related Work

There have been several other works on both alternative languages for the Epiphany, and running high-level languages on co-processors and manycore architectures. In this section, some of the most relevant works will be covered and compared to what is presented in this thesis.

1.3.1 Compute-Focused Languages or Frameworks

Below, some related works that are focused on number-crunching are listed. This contrasts with general-purpose solutions, such as the one this thesis will present, by being more efficient, but more specialised. By necessity, host and accelerator code cannot be written in the same language, although accelerator code might be runnable on the host processor as well as an accelerator.

- **OpenCL**
  A widely supported framework and C-based language. An implementation for Epiphany is provided [1] by commercial actor Brown Deer Technology.

- **Merge**
  Merge [22] is a framework and C++ Domain-Specific Language (DSL). Provides scheduling between CPU and accelerators, spreading load to each processor’s capability. Although superficially the compute kernels might appear to be C++, the DSL is restrictive and prevents general-purpose programming. There is no known implementation for Epiphany.

- **OpenMP[5]**
  OpenMP is an API for parallelising sequential code written in C, C++, or Fortran. The sequential code is annotated with directives as to how it should be parallelised, and an OpenMP capable compiler will parallelise
the code. It is typically used with Symmetric MultiProcessing systems and computer clusters, although it does provide constructs [5, sec. 2.10] for utilising accelerators. Agathos, S. N., Papadogiannakis, A., and Dimakopoulos, V. V. have created [8] an implementation for the Parallella.

While OpenMP does allow the programming of an accelerator in the same language as its host, there is still significant complexity involved in making larger programs run efficiently and race free.

• CAL[15]

CAL is an actor-based dataflow language. Dataflow languages express computation as a series of transformations on pieces of data, called tokens. CAL actors are composed statically, rather than created and destroyed on-demand like Erlang actors. Although CAL is unquestionably a high-level language, dataflow languages are not typically considered general-purpose languages, and are generally used for parallel computation and data processing. For example, the Moving Picture Experts Group (MPEG) standardisation working group uses CAL for the reference implementations of some of their video codecs[10].

A master’s thesis [25] by Mingkun, Y. adapted the C backend of the ORCC CAL compiler to be used with the Epiphany architecture.

• Ambric Architecture and aJava[14]

Ambric is another manycore architecture. Although it sports a communication mesh somewhat comparable to the Epiphany NoC, routing is static, and it exposes an actor-centric message passing programming model instead of a shared-memory one. Message-passing channels need to be configured and allocated in advance. Notably, it can be programmed with a high-level language called aJava, a custom actor-based dataflow language with a syntax reminiscent of Java.

Gebrewanhid et al. [17] presents a CAL compiler that targets both general-purpose CPUs, Epiphany, and Ambric chips by transpiling to C and aJava, respectively.

A master’s thesis [23] by Lindström, J and Nanneson, S uses this compiler to do Epiphany-accelerated MPEG-4 video encoding on the Parallella, but struggles to achieve real-time speeds, or speedups over just using the ARM.

• Epiphany Python

Epiphany Python [12, 13], and its predecessor Epiphany Basic [11] are toy languages that run on the Epiphany. They are noteworthy due to how similar their runtime is to ParallErlang. They are dynamically typed languages that are compiled to a byte-code by the ARM, and then loaded into the shared memory, where they are executed by an interpreter on the Epiphany. It uses a system call mechanism for some operations, such as IO, string concatenation and math functions, which are executed on the ARM. There are four scalar data types available; None (the unit type), Boolean, Integer and Float. There are also strings and arrays of scalars. Processes do not share memory, but communicate by synchronous message passing. There is a mailbox for each (sender, receiver)-pair, which can
contain a single scalar value. The sender waits until the receiver sets a flag indicating it has received the sent value.

However, these languages are simplistic; activation records are statically allocated, FORTRAN style, dynamic memory allocations are never freed, there are no abstraction mechanisms beyond procedures, and no convenient way of constructing more complex data structures. While they serve their purpose as a convenient way of getting started with programming the Parallella very well, they are not designed — or suitable — for large general purpose programs.

1.3.2 General-Purpose-Focused Libraries

These works present libraries that try to simplify using the Epiphany architecture. These are used together with the Epiphany SDK [6], which provides a copy of GCC with an Epiphany backend and C and C++ frontends. A port of the Newlib C standard library implementation is included, as is a library e-lib which provides C APIs for most of the hardware functionalities on the Epiphany, as well as implementations of barrier and mutex synchronisation primitives. There’s also a library for use on the host processor called Epiphany Hardware Abstraction Layer (e-hal), it provides loading and running code on an Epiphany co-processor, as well as abstracted access to the region of DRAM reserved for use by the Epiphany, local memory, and registers of the Epiphany cores.

- ErlPiphany[2]

Erlang bindings for e-hal. Actors can be written in any programming language that can target Epiphany and output an executable file. Erlang is not such a language. By necessity low-level, because of the design of the e-hal. For example, data has to be serialised into binary before it can be sent to (or received from) the Epiphany. The current version of ErlPiphany even requires programs to know the exact absolute memory address for any buffer they want to read from or write to.

- MPI

MPI is a generic message-passing system for use with parallel computers, and is very commonly used on clusters and supercomputers. Richie et al [26] provides an efficient implementation of MPI on the Epiphany architecture.

1.4 Outline

In section 2, “Problem & Project,” the task and expected challenges of this project will be briefly laid out, and in section 3, “Method,” how these challenges were intended to be overcome will be discussed.

In section 4, “A Snapshot View of ParallErlang,” we will see how the implementation looks at the current time, whereas section 5, “Solutions,” will talk in much more detail about how each challenge was solved, and the reasoning behind any major technical decisions.

Finally, some benchmarks trying to gauge the effectiveness of my implementation and optimisations will be presented in section 6, “Results,” and a discussion of these, as well as the entire project, in section 7, “Discussion.”
2 Problem & Project

The Epiphany is configured as an accelerator. It lacks an operating system and the facilities required to run the Erlang Runtime System. Instead, it is intended to receive these facilities from a host processor, an ARM in the Parallella, that runs a typical operating system. As such, it is fundamentally different from anything Erlang has been run on previously, a simple port of the Erlang Runtime System would not work, as it requires the facilities provided by an operating system to run. Running Erlang on an accelerator with a different ISA than its host is a task that has never been attempted before.

A major challenge is to overcome the resource constraints of the Epiphany cores. They are not intended to run big, general purpose programs like the Erlang Runtime System and overcoming these constraints will be critical to the performance of Erlang on Epiphany.

3 Method

3.1 Establishing the Programming Model

In order to solve the problem of not being able to run the Erlang Runtime System in its entirety on the Epiphany, we modified it into a Master-Slave architecture (see Figure 3). The slaves run a stripped down version of the Erlang VM, BEAM, on the Epiphany cores. The master runs on the ARM and submits processes to the slaves. A communication channel is established between the master and the slaves, allowing the master to provide any service to the slaves that they cannot supply themselves.

3.2 Optimising for Performance

However, even a stripped down Erlang VM will not fit comfortably on the Epiphany. In particular, we found that just the main loop of the BEAM emulator compiled to almost 50KB of Epiphany machine code, more than there is memory local to each core. There are a number of solutions to this problem. BEAM could be modified to have fewer instructions, diverging in code and instruction set from the VM in the master runtime system. However, the loader, that converts BEAM bytecode to threaded code, would have to be duplicated and rewritten to generate code for this new VM, which is a significant
undertaking, even disregarding the reduction in code maintainability that comes from duplicating and slightly modifying several modules of the runtime.

Another solution would be to rely on a software implementation of an instruction cache. Since one is provided by the Epiphany SDK [9], it would seem an easy solution. However, in order for the BEAM emulator to benefit from it, it must actually fit in the cache. That means that it has to be broken into several procedures, representing a big change in structure that does not only hamper code reuse, but also register allocation of the variables in the emulator, making it perform worse. It would thus require a significant restructuring of BEAM.

Another solution is to add an Epiphany backend to HiPE [20, 21, 27, 24], the native code compiler in the Erlang Runtime System. If the Erlang code is compiled to machine code, there is no need to fit BEAM into the local memory at all. Instead, only the native code of the Erlang modules in use would need to be in local memory.

The solution we picked was to add an Epiphany backend to HiPE. Since most of the code required for its implementation can be done in Erlang, the burden of implementation is not estimated to be larger than any of the other solutions. It also does not lead to code duplication and bloat in the runtime system, which makes it a much more maintainable solution. Finally, we estimated that this will save the most amount of local memory, since there is no need to store threaded code as well as the machine code implementing an emulator. Instead, all that local memory can be used to store native code of Erlang modules.

4 A Snapshot View of ParallErlang

Largely, the system as implemented is very much structured as it was planned to. BEAM was ported to run on the Epiphany, Erlang processes can be spawned to run on the Epiphany with the epiphany:spawn/1 built-in, and they get regular PIDs and can be communicated with just like any other process. BEAM on the Epiphany performs most operations that require modifying global state by issuing system calls to the Erlang Runtime System on the ARM. Indeed, most of the memory address space that can be accessed by the Epiphany is managed by the ARM, as shown in Figure 4. There are no schedulers running on the Epiphany. Two Erlang processes can thus not time-share on a single Epiphany core, so once all cores are busy, further calls to epiphany:spawn/1 will crash with a system_limit exception. The number of cores available can be discovered with epiphany:count/0.

In Listing 2, we can see an adaptation of the example program from before (Listing 1) adapted for ParallErlang. Note that only start_link had to change.
Listing 2: Erlang statistics module, using ParallErlang

-module(stats).
-export([start_link/0, add_number/2, get_average/1]).

start_link() ->
    Fun = fun() -> loop(0, 0) end,
    try epiphany:spawn_link(Fun)
    catch error:system_limit -> spawn_link(Fun)
    end.

add_number(Server, Number) ->
    Server ! {add, Number},
    ok.

get_average(Server) ->
    Server ! {get_avg, self()},
    receive {avg, Avg} -> Avg end.

loop(Sum, Count) ->
    receive
        {add, Number} ->
            loop(Sum + Number, Count + 1);
        {get_avg, From} ->
            From ! {avg, Sum / Count},
            loop(Sum, Count)
    end.
In this example, we deal with running out of free Epiphany cores by falling back to running the processes on the host processor. Of course, we might want to rewrite the program so that a single process can keep track of multiple sets of numbers, or we might not want to handle the system_limit crash, if we are not going to spawn more processes than there are cores. In that case, letting the program crash when running out would make sure such a condition would not go unnoticed.

A HiPE backend is also available, usable by passing the \{hipe, \{target, epiphany\}\} flag to the compiler. As a first effort to deal with the memory latency problem, there is a simple software implementation of an instruction cache. Any code compiled with HiPE is eligible to be cached, and a function will be cached by a core once it has been called a certain number of times by that core. Eviction only happens when a process dies, and at that point all cached functions are cleared. The call counters, however, are not, so the next time that core calls that function, it will be immediately cached again. Figure 5 illustrates how the memory local to each core is utilised.

5 Solutions

5.1 Establishing the Programming Model

The initial part of the project was to get a stripped down version of the BEAM emulator to run on the Epiphany. The ERTS build system was configured to target the Epiphany, and parts of the runtime system that required external dependencies were torn out. In turn, parts of the system that depended on the torn out modules were also discarded. In the case they were required by the emulator itself, or another critical part of the system, they were instead stubbed out so they compiled, to be touched up later.

5.1.1 Adding SMP Support

When building without SMP support, the Erlang runtime system stores a lot of information about the currently running process in static memory, and would require intrusive refactoring to allow running multiple instances of BEAM in the
same memory space without SMP. Thus, everything that is required to build
the SMP-enabled emulator is needed. For example, several threading primitives;
atomic memory operations, spinlocks, and thread-local storage, are required.

While thread-local storage is easy to provide when threads are cores and
cores can be numbered, spinlocks are trickier since the ones provided by the
Epiphany API only work in SRAM, but the APIs requires their representations
to be embeddable in structures that may be DRAM-allocated. The solution
we implemented is to dynamically allocate native spinlocks from a fixed pool,
representing them as the coordinates of and indices into this pool.

Another operation that is tricky to provide is the memory barrier. While
access to local memory is sequentially consistent by default, accesses going over
the mesh network only promises the bare-minimum level of consistency. There
are no fence instructions or other ways to force sequentially consistent behaviour.
The only synchronisation mechanisms provided are the spinlock and barrier
primitives. The spinlocks will not help as they only protect resources on the core
they are located on and can’t be used in DRAM, and the barriers require all
cores to issue them before any core may complete the barrier operation, which
makes it impossible to use them to provide a memory barrier operation.

If we relax the requirements on the memory barrier operation to only protect
DRAM and make a couple of assumptions about the behaviour of the Epiphany
memory system, we can construct a sequence of instructions implements that
operation. Assume that

1. All memory accesses that take the same route through the mesh network
are strictly ordered.

As mentioned in Section 1.1, there are three mesh networks. One for writes
within a core, one for writes to off-chip resources, such as DRAM, and one
for reads. The manual documents [4, sec. 4.2, p. 20] that writes destined
for the same core are strictly ordered, as are reads destined for the same
core. It is thus reasonable to assume messages on the same mesh network,
taking the same route, cannot overtake each other.

2. All available DRAM is accessed through the same outbound bus.

There are for busses available, east, west, north, and south. On the
currently available Parallella boards, all DRAM is on the east bus, and
this assumption is true.

3. A read following a write to the same address will not finish until the write
has become visible to all other cores.

Since the architecture does not speculate execution, it has no need for a
write buffer. It is thus reasonable to assume a core does not keep track
of the writes it has in flight, especially since there are no documented or
programmer-accessible acknowledgement messages in the mesh network.
However, the manual does promise that “Load operations using data
previously written use the updated values.” [4, sec. 4.2, p. 19] The most
reasonable assumption about the implementation of such a guarantee is
that read messages are not allowed to overtake write messages in the mesh
routers if they are from the same core, and bound for the same memory
address, ignoring the lowest three address bits and the size of the operation.
In that case assumption 3 is true, as long as the destination does not buffer
writes and short-circuits loads from its write buffer.

Then the following sequence will have full barrier semantics:

\[
\begin{align*}
R0 & := \text{load barrier} \\
R0 & := R0 + 1 \\
\text{store barrier, } R0
\end{align*}
\]

```
label:
R1 := load barrier
bne R0, R1, label
```

The branch is used to force the load to be ordered before any instructions
following the barrier, according to the following guarantee from the architecture
reference manual: “Load operations complete before the returned data is used
by a subsequent instruction.” [4, sec. 4.2, p. 19]

Additionally, atomic operations are also required. The Epiphany architecture
only provides test-and-set, and it only works against SRAM. To provide these
operations, we will have to guard the data with a mutex, but as the Erlang
Threading APIs do not require users to deallocate atomic integers, though well-
behaved code always calls `atomic_init` first, the scheme used for the spinlocks is
not usable as-is, because that would eventually lead to running out of spinlocks.
Instead, we use a mutex construction that only requires atomic reads and writes,
and strong memory ordering guarantees: Peterson’s algorithm for \( n \) processes[3,
sec. 2.1.4, p. 22].

**Problems and Solutions** When all dependencies for SMP were resolved, a
strange behaviour appeared. Static data that was initialised during startup was
sometimes going back to zeroed state. It turned out to be a result of the hacky
way the Epiphany toolchain implements the zero-initialised section; rather than
having the loader zero this memory region, the C runtime initialisation code
zeroes this section at program startup. The problem with this is that when
running the same program on multiple cores, one core might have finished this
step and jumped to `void main()`, while another is still zeroing it, leading to
the strange behaviour observed. The solution was to introduce a barrier at the
start of the program, delaying data structure initialisation until all cores have
completed C runtime initialisation.

Furthermore, it turned out that placing the stack in DRAM, as done by the
“legacy” linker script that is part of the Epiphany SDK, caused all of the stacks
to end up at the same address. The simplest solution to the problem was to
modify the linker script to place the stack in SRAM instead.

A problem with placing the C heap in DRAM was also found. Although
newlib libc has interface points for an architecture to provide locking routines to
guard the heap, the Epiphany SDK elects not to use them. Thus, concurrent
allocations from multiple cores sometimes produced the same memory address,
cause subtle errors as Erlang processes had their state overlaid with other
processes’ state. As moving the heap to SRAM was not feasible, and it was
desired to keep ParallErlang using an unmodified SDK, the problem was solved
by introducing wrappers around the `malloc` family that guarded all calls with
a mutex. Since it was not feasible to modify each call to `malloc` to call these
wrappers instead, macros `malloc`, `realloc` and `free` were introduced in the global include file `sys.h` that resolved to the name of their wrapper counterparts.

### 5.1.2 Plugging into ARM ERTS

Once the runtime compiled and ran some trivial hand-crafted threaded code, the next phase was to host it from within an ERTS process on the host. This instance of the ERTS will be referred to as the *master*, as opposed to the instance that is running on the Epiphany, which will be referred to as the *slave*. The address space observed by the Epiphany was mapped in at the same address in the Erlang process on the ARM. Addresses to exported symbols in the Epiphany binary are extracted with `objdump` and written to a header file `slave_syms.h`. This way, many of the tables required to run BEAM could be maintained from the master, and the only code that would need to remain in the slave was the code to read them. A new simple single free-list memory allocator, `slave_alloc`, was introduced to manage shared memory. It was plugged in to the ERTS allocator framework, which allows moving data to use this allocator by just modifying the allocator type declarations in `erl_alloc.types`, without touching any code that actually does the allocations. The shared memory allocator does not use the built-in memory allocator `alloc_util`. In order to combat fragmentation, it keeps the list of free blocks sorted by memory addresses, and allocates short-lived classes of allocations at the lowest possible address and long-lived allocations at the highest.

The atom table was moved to shared memory by using dynamic allocation for the table. The slave is initialised by just updating the table pointer in the slave data section (the address of which is acquired via `slave_syms.h`). `slave_alloc` was modified to forward requests to a fallback allocator in the case that no slave is connected, so that the allocation would work in either case, and automatically place the table in the right location in memory. The slave is only allowed a single operation on the atom table, the only operation that is not guarded by any locks, namely to look up an atom by number. Luckily, that is the only operation that BEAM itself requires (some built-in functions, such as `erlang:list_to_atom/1`, which create new atoms, require more).

The module table contains all currently loaded Erlang modules. A new C module `slave_module` was introduced to manage the module table used by the slave. The utility modules `hash` and `index` were generalised to work with a hash table in shared memory, accessed by different architectures. For example, accessor functions that take the hash function as a parameter, rather than using the function pointer stored in the table, were introduced.

The export table contains entries for all mentioned external functions (module-name-arity-pairs) with a pointer to its entry-point, or a stub that calls the error handler, should the export not be loaded (or not exist). Since some Erlang terms, namely *external funs*, terms on the form `fun Mod:Name/Arity`, contain pointers to export entries, they have to be understandable by both master and slave. They were extended with a second copy of all their fields. Some preprocessor magic makes each runtime call the field it is supposed to use `address`, whereas the other will be named `slave_address` or `master_address`. This way, little existing code needs to change to accommodate this extension. The table and its entries was moved to shared memory, falling back to its original memory allocator with the same mechanism introduced for the atom table.
Access to the module and export table, amongst others, needs synchronisation so that the VM does not intermediary states during hot code loading and upgrades. However, since access to these tables is very frequent, a unique mechanism called \texttt{code\_ix} guards these tables without having readers grabbing mutexes or even memory barriers. Every table guarded by \texttt{code\_ix} contain three copies of itself, although some of them might be empty most of the time. The \textit{active code index}, or \texttt{code\_ix}, is a number between 0 and 2, and decides which of these tables that should be read. When code is loaded or modified, the \textit{staging code index} is assigned to \texttt{code\_ix} + 1 \pmod{3} and the contents of all active tables is copied into the staging tables. Then, code loading proceeds, changing the contents of the staging copies. When loading is complete, the loader waits until all participating threads have passed a checkpoint known as \textit{thread progress}, which means they have issued a memory barrier (and thus are guaranteed to not see stale data from their caches). At that point, \texttt{code\_ix} is updated atomically to the staging index, and the code update “transaction” has been committed. To extend this scheme to the Epiphany cores, a copy of \texttt{code\_ix} was placed in shared memory, and is assigned whenever \texttt{code\_ix} changes. As the Epiphany cores have strictly ordered read accesses, they do not need to participate in the thread progress scheme in order for this synchronisation to work.

Figure 6 illustrates some of the tables used by the BEAM VM, and that the export and atom tables are shared by ARM and Epiphany, whereas they have their own module tables. The BEAM loader, which runs on the ARM, populates all of these tables. In order to do so, it needed to be modified to also be able to generate threaded code for the slave emulator. The BEAM loader module (\texttt{beam\_load.c}) was parameterised on a new type, \texttt{LoaderTarget}, which contains pointers to the opcode tables, the functions used to modify the module tables, offset of the address field in the export entry to use, etc.

In order to feed that code to the slave runtime and to retrieve the opcode table (since it is generated at runtime), a communication channel had to be devised. Communication between the two runtime systems is facilitated through FIFO-buffers in DRAM, where the systems can leave asynchronous messages for each other. There are two FIFO buffers per Epiphany core, one in each direction. In addition, the slave runtime system can perform synchronous syscalls to the master runtime system when it needs assistance before it can proceed. On system startup, the first Epiphany core goes through selected parts of ERTS startup,
sending a SETUP message containing the opcode and built-in function tables. It then signals the other cores in the workgroup to proceed. All cores enter the scheduler loop which performs a READY syscall, which is a request for a process to run. The syscall returns when Erlang code calls the epiphany:spawn family of functions.

Before processes can be spawned on the Epiphany, there is some setup that is required. When Erlang code calls unloaded modules, the aforementioned stub code redirects execution to a special module called error_handler. This module thus has to be loaded before code can be spawned on the Epiphany. In the master runtime system, error_handler is loaded by the module init, which in turn is compiled in to the runtime system, and is loaded during startup. Epiphany does not run init, however. Instead, an Erlang process epiphany_server was introduced, which is started as part of the kernel application, and calls the code module to load the minimal set of modules required to do on-demand code loading. In order to avoid races when spawning Epiphany processes during startup, the epiphany:spawn functions call epiphany_server to get a go-ahead before actually spawning anything.

However, error_handler requires some built-in functions that were stubbed out, such as erlang:whereis/1 and erlang:send/2. Since moving tables that require locks to read to shared memory is undesirable, since it would either require a syscall to lock from the Epiphany, negating most benefits, or require more complex locking algorithms that would slow down operation of the ARM, even when no Epiphany cores are running, and sometimes involve the ARM busy-waiting on the Epiphany, a new approach was needed. A new syscall called BIF was introduced, which copies process state from the process struct on the slave to the process struct on the master, calls the built-in function from the thread that is serving the syscalls, and then copies back the process state and returns. This method is also used to implement erlang:send/2 and several other built-in functions, such as erlang:monitor/2, needed to print text with the io module. Although we would like message passing to be asynchronous, only the case of messaging a PID is without synchronous error conditions that can’t easily be checked from the Epiphany. As a future improvement, that particular case can be made asynchronous.

Receiving messages is a more complicated story. In BEAM, messages are delivered directly into the heap of suspended processes. In particular, the heap is protected by the process main lock, which is held by schedulers running the process. Processes running on the Epiphany are technically always scheduled, but the main lock (in the Process struct on the ARM) is not held, to prevent deadlocks. Instead, the message passing code tests if the recipient is a slave process, and goes into the fallback case for when the process is already running, which is to allocate a standalone heap, a so called heap fragment, to hold the message, and then calls some ParallErlang-specific code for the delivery. It is delivered as a FIFO message, and is linked into the inbox of the process on receipt, just like it would on the ARM. When BEAM dequeues the message, it copies the message to heap, and rather than freeing it with its local dynamic memory allocator, it sends a FREE-message back to the ARM.

That is, however, not all required to do on-demand code loading on the Epiphany. The module that does the heavy lifting when error_handler is called is code_server, which runs on the ARM. Since code is not loaded for the Epipahany automatically when code is loaded for the ARM, code_server needs
knowledge of how code is loaded onto the Epiphany. The invariants that all modules loaded on the Epiphany are also loaded on the ARM, and are the same version, are responsibilities of code_server. To enforce the second invariant, code_server now keeps in memory a copy of the byte code files that it has loaded on ARM, but not yet on Epiphany.

In Figure 7, we can see an example of the communication that happens when running a simple process on the Epiphany. Three threads of execution are drawn. To the left, we have one of the normal scheduler threads on the ARM, in the middle, we have the commander thread, which is the thread running on the ARM that polls for and serves system calls and messages from the slave. On the right, we have an Epiphany core. After system startup, the slave is permanently blocked in a READY syscall that never returns. Once epiphany:spawn is called, the syscall is completed. Note how completion of the system call happens from the scheduler thread, rather than going via the commander thread. Later, the Erlang program running on the ARM decides to send a message to the process on the Epiphany, which uses the MESSAGE message. After copying it to its own heap, the slave replies with a FREE message. Finally, the Erlang process on the slave replies with a message using the BIF system call, and then terminates.

Figure 7: Master-Slave communication when running a simple example process on the Epiphany.
5.1.3 Adding Features Required by More Complex Programs

With dynamic code loading in place, it was now easy to test more complex programs, which in turn uncovered more missing features. Exceptions, for example, required some touch-ups to work. There is a table of exception handler continuation pointers. That table was duplicated in shared memory, so the Epiphany can reach it. Additionally, the table that maps BEAM instruction pointers to function names, used to construct back-traces when no exception handler is found, also needed a touch-up. The table was given a duplicate in shared memory, reusing the same module to maintain both. This made sense since ARM and Epiphany can not execute each others’ BEAM code, so each module loaded need only be inserted into one of these tables. In order for the loader to insert into the correct table, a pointer to the function that inserts into this table was added to the LoaderTarget structure. Another complexity comes from that purging modules from the range tables does not use code_ix. Instead, atomic instructions are used to set the length of those functions to 0, to be deleted from the table next time new code is staged, which does use code_ix for consistency. Luckily reads from the Epiphany are already atomic, and so some simple pre-processor conditionals pick the atomic integer type when compiling for ARM, but just a regular integer when compiling for the Epiphany.

Another feature necessary for more complex programs is garbage collection. To do this, a new syscall GC was introduced. The functionality in the BIF syscall that copies process state between ARM and Epiphany was extracted to its own module slave_state, which could be reused for the implementation for the GC syscall, making it, and the erts_garbage_collect entry point to the garbage collector on the Epiphany, very simple. The garbage collector itself required some generalisation so that it would use the correct memory allocator for the heaps, and not accidentally move the heap of an Epiphany process into memory the Epiphany can’t address. The logic to detect slave processes was introduced to erl_process1h as macros, as to minimise the amount of garbage collection-unrelated logic in the garbage collector.

Many Erlang terms contain pointers to reference-counted data. The first kind of term of this type that was required was the external fun. As mentioned in the previous section, these contain pointers to entries in the export table. In order to construct these in BEAM, we need to be able to increment the reference counters on the table entries. We also need the ability to decrement them when an external fun is received in a message, after the message has been copied to heap (which implies an increment of the reference counter), when we are freeing the heap fragment. This is done on the slave instead of being included in the FREE syscall, because it is closer to what happens on the ARM and allows more code reuse.

However, even though Epiphany has a sufficiently coherent view of shared DRAM, and its word-sized reads and writes are atomic, it is incapable of operations such as atomic increment and decrement. Instead, we implement these operations as asynchronous messages to the ARM, to be executed there. Since we don’t stop and wait for completion (indeed, in that case we would just use a syscall instead), this introduces an ordering requirement between messages and syscalls. Specifically, we must serve all messages from a particular core before we serve a syscall from that core. Otherwise, that syscall might cause a decrement of a reference counter, say, because of a garbage collection, that has...
an increment pending in a message, causing an incorrect release of that resource.

Something else that did not just work was timeouts, when a process on the epiphany wants to sleep for a certain number of milliseconds, possibly cancelling the timeout before it expires, for example, when waiting for a message in a receive statement. These are implemented through inserting a callback in the runtime-global timer wheel, which wakes the process and reinserts it into a scheduler run queue. For the Epiphany, we introduced a new message TIMER that can set and cancel a timeout timer. If the timer expires without a TIMER message to cancel it, a TIMEOUT message is sent back to the Epiphany, where the process is awoken. In order to avoid race conditions, each timeout is associated with an integer identifier. Upon receipt of a TIMEOUT message, the Epiphany first checks that the identifier is associated with the timeout it created last, and that the timeout has not been cancelled.

5.1.4 Attempts at Idling Cores Waiting for Messages

A task that turned out surprisingly difficult was to construct the Event synchronisation primitive using the IDLE Epiphany instruction that places a core in an interruptable sleep. This is desirable to keep cores that are waiting for something from causing unnecessary memory load and power draw. A first approach seemed promising in the functional simulator, but turned out not to work on real hardware. An additional five different approaches were attempted, all of which turned out to be racy in implementation without any obvious reasons for their failure.

The first approach was to use the FSTATUS special register, which allows writing both the IDLE flag and the uninterruptable flag. Using it, both the IDLE flag and the uninterruptable flag can be cleared as a single operation, atomic w.r.t. interrupts. If this operation would actually make the core interruptable and either put the core to sleep, or immediately handle a pending interrupt, it could have been used to implement the wait operation. For example, by checking the flag of the Event in uninterruptable mode, and then using said idle-and-enable-interrupts operation. However, it turned out that writing the IDLE flag does not cause the real hardware to suspend; execution just continues as if nothing has happened.

The first attempt that seemed to work used a very simple interface. On the Epiphany, only two methods were available:

void slave_event_clear(void) Called before checking the inbox FIFO for messages. Clears the Event of the current core.

void slave_event_wait(void) Called after checking the inbox FIFO for messages and finding nothing. Waits on the Event of the current core.

On the ARM, there was only one method:

void slave_event_set(int core_index) Issues a store-store memory barrier, ensuring the FIFO update is visible, and then sets the Event of the core with the given index.

These primitives were used in the following way:

/* Process sleep loop on the Epiphany */
while(1) {
  slave_event_clear();
  check_for_messages();
  if (was_awoken_by_any_message()) break;
  slave_event_wait();
}

/* Syscall wait loop on the Epiphany */
while(1) {
  slave_event_clear();
  if (syscall_finished()) break;
  slave_event_wait();
}

/* On the ARM */
write_fifo_or_complete_syscall(core_index);
slave_event_set(core_index);

The mechanism by which this first attempt sets the event was to interrupt the Epiphany, and inside the interrupt handler replace the IDLE instruction in the wait() method with a NOP. It was hypothesised that when interrupted during wait(), the program counter would either be before the IDLE, and it would be flushed from the pipeline because of the interrupt, and re-decoded after the interrupt returns, or the IDLE would have put the core to sleep, to be immediately awoken by the incoming interrupt. While this approach performed as expected during initial testing, running longer, more complex programs sometimes ended up deadlocking with a core blocking in wait(), while also having a message waiting in its FIFO. Since no bugs could be found in the implementation of the Event, it was assumed that the problem lay in how the Event was used. Since the Event was used to wait for syscall completion in addition to waiting for messages, there was a hypothetical case that could misuse the Event. Because messages were processed inside the sleep loop, it was thought that during processing of a message, a syscall might happen, clearing the event in the process. This does not only seem to not be the case, retroactively, but, if used as written above, the event would always be set after a syscall, since the wait() function does not clear it. Nevertheless, the API of the subsequent attempts was extended to allow for multiple Events, even though there is no way to attach an argument indicating the Event we wish to set, to Epiphany interrupts.

The solution was to design the API around an abstract stack of Events, where the set() operation now sets all the Events on the stack. The interface that all the subsequent implementations shared, was the following:

void slave_event_push(void) Called before checking the inbox FIFO for messages or starting a syscall. Pushes a clear event on the top of the Event stack.

void slave_event_clear(void) Called before rechecking the inbox FIFO for further messages. Clears the topmost Event on the Event stack.

void slave_event_pop(void) Called after exiting a wait loop. Pops the topmost Event off of the stack.
**void slave_event_wait_pop(void)** Waits on the topmost Event on the stack, and then pops it.

**void slave_event_set(int core_index)** Issues a store-store memory barrier, ensuring the FIFO update is visible, and then sets all the Events in the Event stack of the core with the given index.

These primitives were used in the following way:

```c
/* Process sleep loop on the Epiphany */
while(1) {
    slave_event_push();
    do {
        slave_event_clear();
        check_for_messages();
    } while(any_message_was_found());
    if (was_awoken_by_any_message()) break;
    slave_event_wait_pop();
}
slave_event_pop();
```
/* Syscall wait loop on the Epiphany */
while(1) {
    slave_event_push();
    if (syscall_finished()) break;
    slave_event_wait_pop();
}
slave_event_pop();

/* On the ARM */
write_fifo_or_complete_syscall(core_index);
slave_event_set(core_index);

The second attempt tried to use the same mechanism as the first. The events were represented as three-instruction functions, stored on a statically allocated stack, which is called to wait on it:

idle ;; 01b2 ;; nop when set
rts ;; 194f 0402
nop ;; 01a2 ;; pads to a multiple of 4

To push and pop, a pointer is incremented or decremented. The interrupt handler simply loops over the stack, setting all IDLE instructions to NOPs.

When this attempt turned out not to work, yet another approach was taken. The event was still represented by the same three-instruction function, but instead of keeping an actual stack of them, there was just one, and the stack was represented by two integers set and top. When set ≥ top, the event is set. Since the information on whether the event is set or not is now duplicated, the relaxation was chosen to allow the event to be set even when set < top, should a set() interrupt happen during clear(). In practice this meant that the interrupt handler had to set set before modifying the Event, and clear() had to clear the event before modifying set, and never setting set to anything less than top − 1, as might happen if it did set := MIN(set, top − 1). This attempt turned out equally fruitless to the previous one.

The fourth attempt was to use the same stack representation, but use the Event representation from the first attempt, which was a C function where the address of the IDLE instruction was exported using labels-as-values and then modified as per before.

The fifth and final attempt was based on the assumption that the self-modification of code made the Epiphany misbehave. Instead, the idea was to, in the interrupt handler, change the interrupt return address to alter the IDLE instruction, should it be inside the wait() function. This way, set and top could be the single representation of the state of the event, and wait() could say if set < top then IDLE without being racy with the interrupt handler.

The reason that none of these methods worked could be some tricky bug or misunderstanding in my implementation, that managed to survive all the scrutiny and rewrites, or it could be that the IDLE instruction is not meant to be used like we were attempting to, and the hardware implementation might be inherently racy with interrupts arriving just as the processor is going idle.

There are approaches to reducing power draw and memory pressure without IDLE-based Events. One is to use the hardware timers to sleep between polls of FIFOs or syscall fields. Another is to use the TRAP instruction. The TRAP
instruction places the core in an *uninterruptable* halt state, and requires direct access from an outside agent, in our case that would be the runtime system on the ARM, to resume. It is obvious how this instruction would be used to power down during syscalls. To wait for messages with this mechanism, we can introduce a numbering of messages. Both master and slave would independently keep count of how many messages they have sent and received, respectively. Then, we would just introduce a syscall which is “wait until more than $n$ messages have been sent.” Using TRAP is likely the most power efficient of these approaches, as there will be no extra power consumed by a periodic wakeup.

### 5.1.5 Supporting Binaries

The last Erlang data type to be implemented was the *binary*. In Erlang, a binary is an immutable blob of binary data. Binaries are typically used to hold data received from outside the system, such as from the network or filesystem. Since large binaries are common, binaries larger than a threshold are stored outside process heaps, avoiding expensive copying when sending them between processes. These binaries use reference counting for their memory management.

Although we already had a way to touch reference counters, there was a common pattern used with all the reference counted types:

```c
if (refc_decrement_read(&bin->refc) == 0) {
    bin_free(bin);
}
```

This pattern is problematic because the *refc_decrement_read* function could not be implemented asynchronously; the Epiphany needs to know the post-decrement value of the reference counter before it can continue. To solve this, a new reference counter operation, *deffree*, was introduced, transforming the former into the latter:

```c
refc_deffree(bin->refc, DEFFREE_BIN, bin);
```

This operation can be easily implemented as an asynchronous message when called on the Epiphany. By changing all the binary code to use this pattern, it would work on both the ARM and the Epiphany.

However, in order to support all binary-related BEAM operations, the slave also needs to allocate binaries. Since reference counted binaries should be allocated using the allocators on the master, a new syscall, called *BIN*, was introduced that provided the *bin_alloc()* and *bin_realloc()* functions. Freeing was already done through *refc_deffree()*.

Naturally, a binary has to be allocated in shared memory for the slave to be able to read it. However, since it is common to keep large amounts of data in binaries, it’s undesirable to just move all binaries to shared memory, since that puts a severe restriction on their memory use. Instead, the slave should, before it reads from a binary, check that it is located in shared memory, and if not, use the *BIN* syscall to ask that the binary be *migrated* to shared memory. This operation was introduced as a macro *normalise_proc_bin*, and inserted into the macros *ERTS_GET_BINARY_BYTES* and *binary_bytes*, that are used to access the contents of binaries, so that most code would work without change. On the master *normalise_proc_bin* is only a pointer cast. It’s called _proc_bin because the pointer resides in a structure on the Erlang process heap called a
Process Binary, that is a node in the linked list of reference counted resources that are referenced from the process. `normalise_proc_bin` actually alters this pointer so that migration only needs to happen once.

Once a binary has been migrated, the address of the copy is stored in a field `otherp`, so that if migration is requested multiple times, possibly by different slave cores, only a single copy will ever be returned. To make it work with multiple cores serving syscalls, even though there’s only one thread doing that currently, an atomic compare-and-exchange is used to set `otherp`, guaranteeing that the latecomer will notice, free its copy, and return the first one.

Because binaries in master memory now hold references to binaries in shared memory, these might persist even as the last reference from the slave is lost. Additionally, should a slave process send a migrated binary back, a master process will also hold it alive. These issues waste precious shared memory, but are difficult to address. Were migrated binaries to hold a pointer to the master binary they came from, so that binaries may “migrate back”, there would be a reference cycle, preventing reference counting from working safely.

### 5.2 Optimising for Performance

The first step in trying to fit Erlang code in SRAM was the implementation of a HiPE backend targeting the Epiphany architecture.

#### 5.2.1 Writing the HiPE Backend

Since the Epiphany is quite reminiscent of ARM, writing the backend was a relatively straight-forward task. Some design decisions were nevertheless made differently. Loads of link-time immediates into registers are represented as two different instructions in the Epiphany backend. This allows for accurate instruction re-scheduling, in order to prevent stalls in the interlocked pipeline, although it was not implemented during the timeframe of the project. The selection of calling convention, in particular which registers are “fixed,” always containing some of the state of the process, such as PCB, heap, and stack pointers, were made to maximise the use of halfword-wide instructions, in order to minimise code size. Halfword instructions can only use the first eight registers. Because HiPE does not have any callee-saved registers in its calling conventions and because none of its register allocators (discounting the LLVM backend) does range splitting, access to locals on the stack is very common. Thus, the stack pointer was fixed to `r6`. Since heap allocation is common in Erlang code, the heap pointer was fixed to `r7`. In the calling convention we also require the FPU mode to be set to “signed integer” on function entry and exit. Because Erlang only exposes 64-bit floating point operations, we have no use for the 32-bit floating point of the Epiphany. This choice allows us to do integer multiplication without the long save and restore sequences the C compiler emits.

However, because multiplication must never truncate in case of overflow in Erlang, the multiplication instruction is less useful than expected. On Epiphany, the multiplication instruction does not set any flags on overflow, so in order to detect overflow accurately, 64-bit multiplication must be simulated and the result should be tested for overflow. An initial implementation attempted to over-approximate the overflow condition by testing that the upper half of both operands was zero, but that implementation was plagued by bugs. The
implementation in ParallErlang now takes a simpler route and does not inline the '*' operator at all, deferring to the same C implementation as BEAM uses to do the multiplication. A similar complication exists for addition and subtraction, too. Fortunately, the overflow flag does exist for those operators, there just is no instruction that branches when signed overflow happens. Instead, the following sequence is emitted to do addition and then test the overflow flag accessible through the seventh bit of the status special register.

```assembly
  add r0, r1, r2
  movfs r3, status
  lsl r3, r3, #31 - 7
  ble overflow
```

In order to support access to binaries, HiPE code also needs to call the function `normalise_proc_bin`. Luckily, there is an equivalent to the macros `ERTS_GET_BINARY_BYTES` and `binary_bytes` in the HiPE code generator; `hipe_rtl_binary_match:get_base/2`. `normalise_proc_bin` was introduced as a primop `bs_normalise_pb`, and, when targeting a slave, is called in the code generated by `hipe_rtl_binary_match:get_base/2.`

### 5.2.2 Implementing Epiphany Support in the HiPE Runtime System

In order to actually run the code produced by the Epiphany backend, target-specific parts of the HiPE runtime system, such as relocation patching, had to be implemented for Epiphany. In addition, some restructuring was required to be able to target two architectures at the same time. Built-in functions that are used by the HiPE loader (Erlang module `hipe_unified_loader`) that needed it were parameterised on `mode`, an atom, either `master` or `slave`. Since BIFs may only have up to 3 arguments, some of them needed their arguments clumped into tuples to fit.

Some parts of HiPE also needed tweaking to support running in slave mode. When HiPE needs to increment a reference counter, it just calls the threading library’s increment function for atomic integers, since they have for a long time been the same operation but with ParallErlang’s changes to how reference counters are modified, that is no longer true. The HiPE primitive operation `atomic_inc` was renamed `refc_inc`, and now calls the proper function to increment reference counters on either master or slave.

HiPE knows the layout of internal runtime data structures through an auto-generated header file `hipe_literals.hrl`, which in turn defines macros that query a built-in function to retrieve values current to the emulator it is running on. For example one macro defined in this file is:

```erlang
-define(P_HP, hipe_bifs:get_rts_param(22)).
```

Since these offsets may differ between master and slave runtimes, this BIF needed to be parameterised on the runtime system that is targeted, preferably without adding arguments to all of these macros since that would mean big changes to a lot of modules all over HiPE. Luckily, HiPE keeps its target in process dictionary already, so another key `hipe_target_rts` was added, and the above macro was updated as follows:

```erlang
-define(P_HP, hipe_bifs:get_rts_param(22, get(hipe_target_rts))).
```
5.2.3 Working Around Epiphany Memory Bugs

A problem appeared that behaved very much like a compiler bug, but when debugged, turned out to be caused by code sequences that saved a value to stack (which was stored in DRAM), and very quickly restoring that value. The restore would not return the value saved, but an older, stale, value. In essence, the architecture was not providing the basic memory consistency promised by its manual. We had previously observed this problem in C code too, but since the C stack was placed in SRAM, it was never frequent enough to reveal its cause.

As a workaround, the HiPE stack was relocated to SRAM too. This means that any code that is HiPE-compiled has a hard limit on the recursion depth it can use before running out of stack, like a C program would. As a future extension, the HiPE stack could be segmented, storing the currently used segments in SRAM and the higher up segments in DRAM. This would allow recursion as deep as would fit into DRAM, as is expected of an Erlang runtime.

However, as the earlier C problems showed, just moving the stack will not completely fix this problem. For C, we can force an optimisation (`-fgcse-las`) that detects store-load pairs to the same address and omits the load. This is far from a perfect workaround since the optimisation can’t see past function call boundaries (unless they are inlined), but it does fix all previously observed bugs caused by this behaviour.

As for HiPE-compiled code, the workaround will be more complex. A store followed by a function call or return and then a load is very common behaviour in such code, so adding the above optimisation to HiPE is not expected to fix the problem, especially not once some HiPE code runs from SRAM. Instead, we find that, with some adjustments to the assumptions, the memory barrier construction from Section 5.1.1 still work in practice and does guarantee the new value back from heap loads. So, assumption 3 is adjusted as follows:

3. **A read following a write to the same address will not return the new value until it has become visible to all other cores.**

Now, the loop in the barrier sequence becomes useful as more than just a way to introduce a use of the loaded value. We now expect the branch to be taken in some cases. It also becomes important for the correct functioning of the barrier that the new value written is different from what is already stored in memory.

To make HiPE safe under these conditions, we have HiPE-compiled code issue a barrier whenever it constructs something on or modifies the heap. Since this is a common occurrence, we inline this operation. Additionally, we place the barrier value in the PCB, reachable by offsetting the PCB pointer which is already reserved to a register, P. We also reserve register r33 to hold the current value of the barrier, naming it `HFV`, shortening the sequence by one instruction. In assembly, it becomes:

```assembly
add   HFV, HFV, 1
str   HFV, (P, #P_HEAP_FENCE)
retry:
1dr   tmp, (P, #P_HEAP_FENCE)
sub   tmp, tmp, HFV
bne   retry
```
5.2.4 Implementing a Software Instruction Cache

It is expected, because of how the architecture is designed, that a lot of potential performance is lost by running code from DRAM. However, with a HiPE backend for Epiphany, we can now adjust the linking of that machine code as we see fit, including directing function calls through a dispatch table with an entry per core that allows us cache functions in SRAM; essentially implementing a software instruction cache.

As a first step, the portion of the linker that is implemented in Erlang, in a module called \texttt{hipe\_unified\_loader}, was refactored so that functions were linked individually. This allowed for, in the HiPE runtime system, placing all trampolines next to the function code, creating a unit of position-independent code. The trampolines, which use absolute addressing, were typically emitted when a branch from the HiPE-compiled code, using relative addressing, could not reach its target. Now, trampolines are unconditionally emitted for each target, since any relative branches out of a function would prevent that function code from being copied somewhere else.

Next, an optional interface-point was introduced for HiPE targets to generate trampolines and dispatch-tables, also associating them with the stack descriptors belonging to the function. The feature is exposed to the loader as a builtin \texttt{hipe\_bifs:cache\_insert/2}. The entry emitted looks like this:
struct fun_entrypoint {
    UWord code[4];
    void *cold_address;
    Uint size;
    struct sdesc **sdescs;
    Uint sdesc_count;
    struct {
        void *address;
        Sint count;
    } table[];
};

The field table is the dispatch table, and initially, each entry points to an assembly routine hipe_cold_call. code is the actual trampoline. It loads the address of table into an unused register, and then jumps to the address entry corresponding to the core it is running on. On Epiphany, it looks like this:

```
    mov r12, %low(ADDRESS_OF_TABLE)
    movt r12, %high(ADDRESS_OF_TABLE)
    ldr r0, [r12, COREIX]
    jr.1 r0
```

COREIX is added as yet another reserved register, r34, permanently containing the index of the current core, times eight. r12 is chosen to contain the dispatch table address since it is the intra-procedure scratch register in the C calling convention, but any caller-save register not used for argument passing could be used in its stead.

hipe_cold_call uses the address of table in r12 to find the other fields in struct fun_entrypoint. It increments the count for the current core. Should it exceed a hard-coded threshold, and there is also enough remaining space in the cache, it will call a C routine to perform caching. Otherwise, it jumps to cold_address.

The caching routine receives the address of table and the value of COREIX as arguments. It allocates space in the cache by subtracting from the field containing the amount of remaining space, and copies the function code into the cache. It also appends an entry to a table of cached functions on the particular core. Finally, it updates the dispatch table to point to the copy of the function in the cache, performs a fence, and returns the new address.

For simplicity, eviction only happens when an Erlang process terminates. Because of that, eviction needs only reset the dispatch tables and counters modified. Otherwise, eviction would either require a stack walk to touch up any return addresses pointing to the code that is being evicted, cached entries to be reference-counted, or return addresses also redirected through dispatch-tables too; increasing complexity and the amount of DRAM-accesses performed by hot code.

As for other stack walks, such as for exception handling and garbage collection, the return addresses to hot code will not be found in the global hash table of return addresses, nor could they be inserted, since the table is shared by all cores. Instead, the routine that makes stack descriptor lookups was extended to, if nothing is found in the hash table, perform a binary search over the core-specific table of cached functions to find the correct function, and then compute the corresponding cold return address to look for instead.
6 Results

In order to measure the performance of ParallErlang, as well as the benefits of our optimisations, we have adapted a set of benchmarks from the Computer Language Benchmarks Game[16] to use the Epiphany. Additionally, a custom benchmark was written doing simple blocked parallel matrix multiplication of lists-of-lists-of-integers. Integer math was used since floating-point math is a weak point of the Erlang Runtime System, Epiphany lacks hardware support for double precision floating-point math (and so the same C routine will be used from both BEAM and HiPE), and another benchmark (mandelbrot) already covers floating-point math. The input sizes to the benchmarks were tuned to make them finish in reasonable time on the Epiphany. The matrix multiplication benchmark additionally had the values in the input matrices tuned down to decrease the amount of bignums in the output, and it had the block size tuned to maximise performance on the Epiphany. Sources of the benchmarks can be found in Appendix A.

We use a benchmarking script (See Appendix A.5.1) that makes sure all required modules are loaded, and all Epiphany-related startup has happened before running the benchmarks. It also allows benchmarks to pre-generate data (such as the matrices to be multiplied) before the measurements are taken. Each benchmark is run ten times and the runtime is recorded. Additionally, we record the total number of reductions, the number of Erlang function calls, when running on the ARM. For processes running on the Epiphany, we instead capture the results of three performance counters; total cycles, cycles spent stalling for instruction fetches, and cycles spent stalling for data fetches. However, since only two performance counters can be used at a time, we use the first one for total cycle count, and interleave the other one between fetch and load stalls. The fraction of cycles spent in the chosen type of stall is the number recorded. Thus, there are only five samples of those two measurements. In order to make these measurements, the performance counters were exposed as a builtin epiphany:timers/2, and the runtime system was adjusted to pause these counters while a process is waiting for system call completion or messages.

We run this benchmark script both on BEAM-files that have not been compiled with HiPE (and should reflect the performance prior to adding the HiPE backend) as well as files that have. Additionally, we also run them on the branch where the software caching is implemented. ARM results are included for this branch too since the alteration to HiPE-code loading might affect the results. An important point to note about the benchmarks of the software caching is that the call counts are not cleared between samples. Since the number of previous calls required to qualify a function for being cached is four, the number of cached functions increases after the first four samples, as all functions that are eligible are now cached. These results are displayed in Figures 8 and 9.

Additionally, the benchmarks that can pick a number of cores to use without altering the workload size are also measured on all power-of-2 numbers of cores to measure how well the benchmarks and the implementations scale. This benchmark was ran ten times on the ARM, but only twice on the Epiphany, because of how long-running it was. These results are displayed in Figures 10, 11, 12, 13, and 14.

Finally, energy efficiency estimations, based on the maximum rated power consumptions (as listed in Section 1.1) of the respective processors, in Erlang
Figure 8: Performance data running four benchmarks on ARM, noting means and standard deviations.
(a) Runtime of benchmarks on Epiphany

<table>
<thead>
<tr>
<th>benchmark</th>
<th>$\mu_{\text{time}}$ (s)</th>
<th>$\sigma_{\text{time}}$</th>
<th>$\mu_{\text{cycles}}$</th>
<th>$\sigma_{\text{cycles}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees</td>
<td>39.01</td>
<td>0.23</td>
<td>$5.34 \cdot 10^{10}$</td>
<td>$1.88 \cdot 10^{8}$</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>27.35</td>
<td>0.3</td>
<td>$2.27 \cdot 10^{11}$</td>
<td>$1.58 \cdot 10^{8}$</td>
</tr>
<tr>
<td>matmul</td>
<td>11.81</td>
<td>$4.73 \cdot 10^{-2}$</td>
<td>$1.07 \cdot 10^{11}$</td>
<td>$1.47 \cdot 10^{8}$</td>
</tr>
<tr>
<td>fannkuchredux</td>
<td>14.18</td>
<td>$5.68 \cdot 10^{-2}$</td>
<td>$5.9 \cdot 10^{10}$</td>
<td>$4.94 \cdot 10^{7}$</td>
</tr>
</tbody>
</table>

(b) Runtimes, total CPU-cycles, fraction of cycles stalling for instructions, and fraction of cycles stalling for data on Epiphany (BEAM)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>$\mu_{\text{fstall}}$</th>
<th>$\sigma_{\text{fstall}}$</th>
<th>$\mu_{\text{lstall}}$</th>
<th>$\sigma_{\text{lstall}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees</td>
<td>0.81</td>
<td>$3.42 \cdot 10^{-4}$</td>
<td>0.29</td>
<td>$3.46 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>0.93</td>
<td>$5.33 \cdot 10^{-5}$</td>
<td>$6.13 \cdot 10^{-2}$</td>
<td>$2.19 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>matmul</td>
<td>0.78</td>
<td>$3.31 \cdot 10^{-4}$</td>
<td>0.26</td>
<td>$2.1 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>fannkuchredux</td>
<td>0.73</td>
<td>$3.19 \cdot 10^{-4}$</td>
<td>0.33</td>
<td>$2.02 \cdot 10^{-4}$</td>
</tr>
</tbody>
</table>

(c) Runtimes, total CPU-cycles, fraction of cycles stalling for instructions, and fraction of cycles stalling for data on Epiphany (HiPE)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>$\mu_{\text{time}}$ (s)</th>
<th>$\sigma_{\text{time}}$</th>
<th>$\mu_{\text{cycles}}$</th>
<th>$\sigma_{\text{cycles}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees</td>
<td>10.55</td>
<td>$5.3 \cdot 10^{-2}$</td>
<td>$1.77 \cdot 10^{10}$</td>
<td>$5.96 \cdot 10^{8}$</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>23.61</td>
<td>$2.97 \cdot 10^{-2}$</td>
<td>$2.12 \cdot 10^{11}$</td>
<td>$1.33 \cdot 10^{8}$</td>
</tr>
<tr>
<td>matmul</td>
<td>6.52</td>
<td>0.1</td>
<td>$5.57 \cdot 10^{10}$</td>
<td>$3.07 \cdot 10^{8}$</td>
</tr>
<tr>
<td>fannkuchredux</td>
<td>7.57</td>
<td>$2.43 \cdot 10^{-2}$</td>
<td>$3.18 \cdot 10^{10}$</td>
<td>$8.29 \cdot 10^{6}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>benchmark</th>
<th>$\mu_{\text{fstall}}$</th>
<th>$\sigma_{\text{fstall}}$</th>
<th>$\mu_{\text{lstall}}$</th>
<th>$\sigma_{\text{lstall}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees</td>
<td>0.84</td>
<td>$2.19 \cdot 10^{-3}$</td>
<td>0.16</td>
<td>$5.77 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>0.94</td>
<td>$4.69 \cdot 10^{-5}$</td>
<td>$3.74 \cdot 10^{-2}$</td>
<td>$1.26 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>matmul</td>
<td>0.84</td>
<td>$8.57 \cdot 10^{-4}$</td>
<td>0.16</td>
<td>$1.9 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>fannkuchredux</td>
<td>0.82</td>
<td>$8.26 \cdot 10^{-5}$</td>
<td>0.23</td>
<td>$3.75 \cdot 10^{-5}$</td>
</tr>
</tbody>
</table>
### Table 1: Energy efficiency in Erlang function calls (reductions) per second and Joule when running the benchmarks on ARM and Epiphany

<table>
<thead>
<tr>
<th>benchmark</th>
<th>ARM (HiPE)</th>
<th>Epiphany (Caching)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>reds/s</td>
<td>reds/J</td>
</tr>
<tr>
<td>binarytrees</td>
<td>1.27 · 10⁷</td>
<td>4.27 · 10⁶</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>1.16 · 10⁶</td>
<td>3.85 · 10⁵</td>
</tr>
<tr>
<td>matmul</td>
<td>8.05 · 10⁶</td>
<td>2.68 · 10⁵</td>
</tr>
<tr>
<td>fannkuchredux</td>
<td>7.31 · 10⁶</td>
<td>2.44 · 10⁶</td>
</tr>
</tbody>
</table>

---

**Figure 9:** Performance data running four benchmarks on Epiphany, noting means and standard deviations

**Figure 10:** Strong scaling of benchmarks on ARM (BEAM)
Figure 11: Strong scaling of benchmarks on ARM (HiPE)

Figure 12: Strong scaling of benchmarks on Epiphany (BEAM)
Figure 13: Strong scaling of benchmarks on Epiphany (HiPE)

Figure 14: Strong scaling of benchmarks on Epiphany (HiPE w/ Caching)
reductions (function calls) per Joule of electrical energy are shown in Table 1. Because of bugs in ParallelErlang, the number of reductions reported for processes running on the Epiphany are inaccurate. As the benchmarks run the same code on Epiphany and ARM, the reasonable assumption that the benchmarks consumed the same number of reductions on both processors was made.

7 Discussion

7.1 Performance

As we can see in Figure 9a, the use of HiPE with our Epiphany backend radically increases the performance of ParallelErlang. We also see that the introduction of the software cache further increases performance, although by a smaller amount.

One immediate observation is how poorly ParallelErlang performs on Epiphany when compared to ERTS on the modest ARM on the Parallella. If we compare the runtimes on Epiphany using HiPE with software caching (Figure 9d) with the ones on ARM using HiPE (Figure 8c), we note that binarytrees takes 100 times as long running on the Epiphany as it did on the ARM. Since the benchmark contains a large sequential section, it was expected to run significantly slower on the Epiphany. However, even the matmul benchmark, which is not only fully parallelised, but also tuned to run optimally on the Epiphany, runs 25 times slower than on the ARM. Although not unexpected, it is still striking, especially compared to the numbers of super-optimised close-to-theoretical-optimum C benchmarks like the scalable matrix multiplication algorithm for Epiphany that Adapteva published[28].

Another point to note is the performance gained by the introduction of software caching. Looking at Figure 9, we see that the performance gain was 33% in the binarytrees benchmark, and even less in the others, raises the question on whether the assumption that running code from shared memory was the primary bottleneck was wrong, or if the time spent in the runtime system, which is never cached, was simply underestimated. The mandelbrot and fannkuchredux benchmarks are the two benchmarks out of the four that call a built-in function as part of their innermost loops, and that seems to correspond with very marginal gains from the software instruction cache. In order to answer the question of why the performance gains are not larger, the performance counters that measure numbers of cycles wasted on stalling for DRAM accesses were added to the benchmark (Table 1). Interestingly, and counter to intuition given the descriptions of the performance counters, several of the benchmarks show the two stall counters summing to more than the total number of cycles. Curiously, this is most pronounced when using BEAM. There are a couple of possible explanations for this:

a) The result is a random error. This seems unlikely, since the sum is more than 10 standard deviations greater than 1 in the binarytrees on BEAM case.

b) There is a systematic error in the benchmarks, since the two counters were not sampled simultaneously. The only known systematic error is the lack of call-count reset in the software cache. That implies that the error should only be present when the software cache is enabled. However, in the
results, the error is not only similar between the HiPE and Caching cases, but also it is the greatest using BEAM. This is thus a poor explanation.

c) The “stall for external instruction fetch” (fstall) counter does not only increment when the core has to be fully stalled to await an instruction fetch, but also when the decoder only has one instruction to consider, instead of the maximum current dispatch, two. Additionally, since a “stall for external load” (lstall) does stall the entire core, and thus provides a window for the instruction fetch to catch up, it is curious why the fstall counter would be increased in that case.

In this case, the fstall number will be inflated, and it is hard to tell by how much. It would, however, still be a lower bound on the number of cycles spent executing code from external memory.

Assuming the most reasonable explanation, c, it is notable how large fstall is in all the benchmarks on the software caching branch. It is clear that there is hot code that is never cached in any of the benchmarks. Thus, an explanation to the poor increase in performance could be that the time spent executing (HiPE-compiled) Erlang code was over-estimated, and that there are hot-spots elsewhere, presumably in C code, that require either static allocation to SRAM, or to be included in the software caching solution.

Another very striking result is the cycle counters (Figures 9c and 9d). With the introduction of the software cache, the number of cycles spent on the Epiphany falls by 2-3 orders of magnitude in all benchmarks, which is dramatically different from the reductions in wall-clock time. This might indicate errors in how the performance counters were measured, or, it might indicate that there is some other bottle-neck that is causing the Epiphany cores to spend most of their time idling. One guess as to what that would be is the system call latency, due to the lack of interrupts. Since all the benchmarks do heap allocation in their inner loops (it is true of mandelbrot because floats are boxed), it is reasonable to expect that they require frequent garbage collections. It is feasible that the majority of time is spent waiting for the ARM to wake up and garbage collect all the heaps. Since it would be quite possible to do garbage collections from the Epiphany, as long as the heap does not need to be enlarged, further experiments are needed to investigate the reason for these results.

Note also the high standard deviations in the cycles measurements when using software caching. This is most likely explained by the lack of call-count resets, as after four samples, the benchmarks change with more functions are being inserted into the cache.

We can see in Figure 8, although it is within margin for error in the individual benchmarks, that there might be a very slight performance regression for ARM in the caching branch. This is not unsurprising considering that functions from the same module have been forced to call each other via trampolines, and risk being loaded far apart, missing out on cache locality benefits.

The matmul benchmark show the least difference between ARM and Epiphany, and additionally show poor scaling on the ARM (Figure 11). This is likely due to the block size being tuned lower than optimal for something with a few fast cores; and should the block size have been tuned to the different targets individually, the performance difference would likely be similar to that exhibited by the other benchmarks.
The energy efficiency estimations (Table 1) confirm that there is still work to do before ParallErlang can make use of the energy efficiency improvements brought by the Epiphany architecture. Since the two processors have roughly the same rated power draw (3W for the ARM vs 2W for the Epiphany), this was not very surprising given the run times. However, there are several points that needs to be considered when comparing these numbers. First, two of the benchmarks, binarytrees and fannkuchredux, do not load all 16 cores for the duration of the benchmark, which skews the results since those cores are still counted as if they were operating at full power. Second, the Epiphany-III is manufactured using a 65nm semiconductor manufacturing process which is inherently less energy efficient than the 22nm process used for the Zync.

7.2 Future work

The software instruction cache for HiPE needs to be finished. In addition to addressing the regressions it causes, HiPE should also be adjusted so that recursive calls can be call-counted. This is the primary reason that caching thresholds are so small (such as 4) in the current prototype, and by increasing it, bigger programs should be able to take advantage of it without the cache filling up with rarely called functions.

Another task that is overdue is to implement idle waiting using TRAP as detailed in the end of Section 5.1.4. As the reason for this work was energy efficiency, such an easy and important improvement as not having cores busy-waiting at full power when they are unused should be prioritised.

As the results seem to indicate that a large proportion of time is spent executing code from external memory, even when using the software instruction cache, a profiler should be introduced that can reveal such hot-spots, allowing moving them to local memory.

The HiPE stack should be segmented, with only the top segments stored in local memory. This would allow the amount of local memory used by the HiPE stack to be decreased, freeing up this precious resource to be used elsewhere.

Another obvious improvement is to use local memory for the heaps of Erlang processes, especially when local memory is freed up from the HiPE stack segmenting. Presumably, once slowdowns due to instruction fetch from external memory is dealt with, the bottleneck is going to become external data fetches instead. Of course, it is not acceptable to impose a strict maximum heap size on Erlang programs, so the heaps must be allowed to outgrow the local memory. A suggestion would be to first expel the long-lived heap to shared memory, and later, if required, also the nursery, possibly repurposing that memory for other uses, such as more software cache. Also, minor collections should be done from the Epiphany, at least while the heap lives in local memory.

7.2.1 What Can Be Done With a Newer Model Epiphany

It is expected that a newer model Epiphany would have more cores than existing models. In such a scenario, the effort to move all hot code and data to local memory becomes even more important, as the bandwidth to shared memory will only increase by the square root of the number of cores, at best. Thus, when increasing the number of cores, there will be a point where memory bandwidth
becomes the main bottleneck, and performance will scale much worse than linearly from that point.

However, it is not unreasonable to guess that such a model also will come with more local memory. As such it will probably be required to continue the effort of efficient local memory usage and moving of hot memory locations to local memory in order to make full use of such a model.

7.2.2 What Does It Take to Extend to Another Co-processor

In ParallErlang, there are several assumptions made about the co-processor. It is assumed that it is of the same word size as the host. It is assumed that there is (at least the illusion of) some shared memory that both cores can access in a cache-coherent manner. This memory must be mappable into the host’s ERTS process’ memory space at the same address it is visible to the co-processor at. These assumptions allow the host and co-processor to use the same term format, removing the need for any conversions. It is also assumed that there is a fully-featured C compiler, and for performance’s sake, it should support the GCC extension labels as values, so that the directly threaded BEAM can be used. If it does not, some extra tweaking to the loader might be required to emit the correct byte code.

Except for these assumptions, there is not too much of ParallErlang that is Epiphany-specific. The code that sets up the co-processor and memory mapping needs to be adjusted to extend to another co-processor, as does the code that implements system calls. There are some functions that determine whether a memory address is in shared memory, and whether it belongs to the co-processor heap, that do need touch up. Also, there is some debugging code that isn’t required to use ParallErlang, but is nevertheless useful, such as void epiphany_backtrace(), that is Epiphany specific. Finally, all the typical requirements of an ERTS port apply. In particular, an implementation of Ethreads need to be provided since co-processors tend to lack Pthreads. The shim that implements the atomics API on Epiphany can be reused for any architecture with some code changes.

7.3 Conclusions

Although power-efficiency is not yet en par with ARM CPUs, the benchmarks seems to indicate that there might still be plenty of untapped performance. Additionally, bringing the Epiphany to a comparable semiconductor manufacturing process would be expected to narrow the performance gap.

All-in-all the project could be considered a success. The system works and exceeds initial expectations of compatibility, although the limitation on the number of concurrent processes on the Epiphany makes porting a program to ParallErlang far from a push-button exercise.

We’ve now seen how it is possible to port a high-level language to a co-processor in a way that abstracts most of the complexities of programming such a processor from the programmer, and allows them to reuse existing code on more power-efficient architectures.
References


Appendices

A Benchmark Sources

Benchmarks binarytrees, matmul, and fannkuchredux are copyright © their respective authors, and are provided under the Revised BSD license. For brevity, the license is included here once, rather than in the respective source code listings.

Revised BSD license

Copyright © 2004-2008 Brent Fulgham, 2005-2015 Isaac Gouy

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
* Neither the name of "The Computer Language Benchmarks Game" nor the name of "The Computer Language Shootout" Benchmarks nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

A.1 Binarytrees

% The following benchmark is based on an entry in
% The Computer Language Benchmarks Game
% http://benchmarksgame.alioth.debian.org/
% contributed by Isaac Gouy (Erlang novice)
% parallelized by Kevin Scaldeferri
% Copyright © 2004–2008 Brent Fulgham, 2005–2015 Isaac Gouy

-module(binarytrees).
-export([prepare/0, bench/1, main/1]).
-export([depth/2]).

-module(binarytrees).
-export([prepare/0, bench/1, main/1]).
-export([depth/2]).

-define(Min,4).

prepare() ->
    {nil, [colib, timer, lists, io]}.

bench(nil) ->
    main(['"10"']).

main([Arg]) ->
    N = list_to_integer(Arg),
    Max = lists:max([?Min+2,N]),
    
    Stretch = Max + 1,
    io:fwrite("stretch_tree_of_depth_w\tcheck:\w-n",
        [ Stretch, itemCheck(bottomUp(0,Stretch)) ]),
    
    LongLivedTree = bottomUp(0,Max),
    depthLoop(?Min,Max),
    
    io:fwrite("long_lived_tree_of_depth_w\tcheck:\w-n",
        [ Max, itemCheck(LongLivedTree) ]),
    
    ok.

depthLoop(D,M) ->
    Results = colib:pmap({?MODULE, depth}, [M], lists:seq(D, M, 2)),
    lists:foreach(fun(Result) ->
        io:fwrite("-\w\t\trees_of_depth\w\t\check:\w-n", Result)
    end, Results).

depth(D,M) ->
    N = 1 bsl (M-D + ?Min),
    [ 2*N, D, sumLoop(N,D,0) ].

sumLoop(0,_,-Sum) -> Sum;
A.2 Mandelbrot

% The following test is based on an entry in
% The Computer Language Benchmarks Game
% http://benchmarksgame.alioth.debian.org/
% Contributed by Johan Karlsson based on Fredrik Svahn's
% mandelbrot program
% Copyright © 2004-2008 Brent Fulgham, 2005-2015 Isaac Gouy

-module(mandelbrot).
-export([prepare/0, bench/1, bench/2]).
-define(LIM_SQR, 4.0).
-define(ITER, 50).
-define(SR, -1.5).
-define(SI, -1).
-define(DEFAULT_WORKERS, 16).

prepare() -> [nil, [lists]].

bench(nil) -> bench(nil, DEFAULT_WORKERS).
bench(nil, NoWorkers) ->
  main(["64"], NoWorkers).

main([Arg], NoWorkers) ->
  N = list_to_integer(Arg),
  Jobs = lists:seq(0, N-1),
  Self = self(),
  Row = fun(Y)-> Self ! {Y, row(N-1, 0, ?SI+Y*2/N, N, 0, [], 7)} end,
  spawn_link(fun() -> workerserver_entry(Row, Jobs, NoWorkers) end),
  [
    P4\n  , Arg, ",", Arg, \"\n\"] ++ [receive {Job, C} -> C end || Job <- Jobs].

workerserver_entry(Fun, Jobs, NoWorkers) ->
  Self = self(),
  {Spawn, _Count} = colib:spawn_and_count(),
  Workers = [Spawn(fun() -> worker(Fun, Self) end)
|| _ <= lists:seq(1, NoWorkers)],
workserver(Workers, Jobs).

workserver(Workers, [Job|Jobs]) ->
  receive {get_job, Worker} ->
    Worker! {job, Job},
    workserver(Workers, Jobs)
  end;
workserver(Workers, []) ->
  lists:foreach(fun(W) -> W ! stop end, Workers).

worker(Fun, Jobserver) ->
  Jobserver! {get_job, self()},
  receive
    stop -> ok;
    {job, Job} ->
      Fun(Job),
      worker(Fun, Jobserver)
  end.

%% Iterate over a row, collect bits, bytes and finally print the row
row(X, X, _, _, Bits, Bytes, C) ->
  case C of
    7 -> lists:reverse(Bytes);
    C -> lists:reverse([Bits bsl (C+1) | Bytes])
  end;

row(M, X, Y2, N, Bits, Bytes, 0) ->
  row(M, X+1, Y2, N, 0, [Bits bsl 1 + m(?ITER, ?SR+(X+X)/N, Y2)
    | Bytes], 7);
row(M, X, Y2, N, Bits, Bytes, BitC) ->
  row(M, X+1, Y2, N, Bits bsl 1 + m(?ITER, ?SR+(X+X)/N, Y2),
    Bytes, BitC-1).

%%%Mandelbrot algorithm
m(Iter, CR, CI) -> m(Iter - 1, CR, CI, CR, CI).

m(Iter, R, I, CR, CI) when is_float(R), is_float(I),
  is_float(CR), is_float(CI) ->
  case R*R+I*I > ?LIM_SQR of
    false when Iter > 0 -> m(Iter-1, R-R-I+CR, 2*R+I+CI,
      CR, CI);
    false -> 1;
    true -> 0
  end.

A.3 Matmul
-module(matmul).
-export([prepare/0, bench/1, bench/2, test/1, mul/3]).

-define(SUBSIZE, 4).
-define(ROWS, 96).
-define(COLS, 96).

prepare() ->
    Pregen = {matgen(), matgen()},
    {Pregen, ![MODULE, lists, epiphany, epiphany_server]}. 

bench(Pregen) -> bench(Pregen, 16).
bench([A, B], NoWorkers) ->
    mul(A, B, NoWorkers).

-type matrix(0f) :: [[0f]].
-type matrix() :: matrix(integer()).

test(N) ->
    [A, B] = {matgen(), matgen()},
    {Time, Res} = timer:tc(fun() -> mul(A, B, N) end),
    case mul_seq(A, B) of
        Res -> Time;
        Correct -> {badmatch, Res, Correct}
    end.

%% @doc Multiply matrices A and B using N workers
mul(A, B, N) ->
    Workers = fork(N),
    try
        As = split_rows(A, []),
        Bs = split_cols(B, []),
        do_mul(As, As, Bs, Workers, Workers, [])
    after
        join(Workers)
    end.

-spec do_mul([matrix()], [matrix()], [matrix()], [pid()], [pid()],
            matrix(reference())) -> matrix().
do_mul([], AllAs, [], _Workers, _AllWorkers, Acc) ->
    collect(Acc, []);
do_mul([], AllAs, [B|Bs], Workers, AllWorkers, Acc) ->
    do_mul(AllAs, AllAs, Bs, Workers, AllWorkers, [[]|Acc]);
do_mul(As, AllAs, Bs, [], AllWorkers, Acc) ->
    do_mul(As, AllAs, Bs, AllWorkers, AllWorkers, Acc);
do_mul([AllAs], AllAs, [B|_]=Bs, [Worker|Workers], AllWorkers,
            [Acc|Accs]) ->
    Ref = make_ref(),
    Worker ! {mul, self(), Ref, A, B},
do_mul(As, AllAs, Bs, Workers, AllWorkers, [[Ref|Acc]|Accs]).

-spec collect(matrix(references()), matrix(matrix()), matrix().
collect([], Acc) -> merge(Acc);
collect([Col], Acc) -> collect(Col, [[|Acc]);
collect([[Row]|Col], [A|Acc]) ->
    receive {Ref, Mat} ->
        collect([Row|Col], [[Mat|A]|Acc])
    end.

%% @doc Flattens a MxN matrix of KxK matrices into a MKxNK matrix.
-spec merge(matrix(matrix())), matrix() -> matrix().
merge([], _) -> [];
merge(Mat) ->
    Heads = [hd(Row) || Row <- Mat],
    Tails = [tl(Row) || Row <- Mat],
    merge_row(Heads, []) ++ merge(Tails).

-spec merge_row(matrix(), matrix()) -> matrix().
merge_row([], Acc) -> lists:reverse(Acc);
merge_row(Row, Acc) ->
    Heads = [hd(Mat) || Mat <- Row],
    Tails = [tl(Mat) || Mat <- Row],
    MergedRow = lists:append(Heads),
    merge_row(Tails, [MergedRow | Acc]).

%% @doc Forks an integer.
-spec fork(integer()) -> [pid()].
fork(N) ->
    {Spawn, _Count} = colib:spawn_and_count(),
    [Spawn(fun worker_loop/0) || _ <- lists:seq(1, N)].

join(Workers) ->
    Refs = [monitor(process, W) || W <- Workers],
    lists:foreach(fun(W) -> W ! stop end, Workers),
    lists:foreach(fun(R) -> receive {'DOWN', R, _, _, _} -> ok
                       end end, Refs).

worker_loop() ->
    receive stop -> ok;
    {mul, From, Ref, A, B} ->
        From ! {Ref, mul_seq(A, B)},
        worker_loop()
    end.

%% Multiplies two matrices
-spec mul_seq(matrix(), matrix()) -> matrix().
mul_seq(ARows, B) ->
    BCols = transpose(B),
    mul_ARows = maplist(fun(Row) ->
                        lists:map(fun(A) -> BRow = [BCol ||
                                                  Col <- A]
                              end, Row)
                        end, ARows),
    lists:append(mul_ARows, []).

53
mul_strips(A, B) -> mul_strips(A, B, 0).
mul_strips([], [], Acc) -> Acc;
mul_strips([A|As], [B|Bs], Acc) ->
    mul_strips(As, Bs, A+B + Acc).

-spec transpose(matrix()) -> matrix().
transpose([[[],]]) -> [];
transpose(Mat) ->
    Heads = [hd(Row) || Row <- Mat],
    Tails = [tl(Row) || Row <- Mat],
    [Heads | transpose(Tails)].

-spec matgen() -> matrix().
matgen() ->
    random:seed(now()),
    [[random:uniform(3) || _Col <- lists:seq(1, ?COLS)]
      || _Row <- lists:seq(1, ?ROWS)].

%% @doc Cuts a (Mx?SUBSIZE)xN matrix into M ?SUBSIZExN matrices.
-spec split_rows(matrix(), [matrix()]) -> [matrix()].
split_rows([], Acc) -> lists:reverse(Acc);
split_rows(Mat, Acc) ->
    {Row, Rest} = lists:spl2t(?SUBSIZE, Mat),
    split_rows(Rest, [Row|Acc]).

%% @doc Cuts a Mx(Nx?SUBSIZE) matrix into N Mx?SUBSIZE matrices.
-spec split_cols(matrix(), [matrix()]) -> [matrix()].
split_cols([], Acc) -> lists:reverse(Acc);
split_cols(Mat, Acc) ->
    Tuples = [lists:spl2t(?SUBSIZE, Row) || Row <- Mat],
    Col = [Row || {Row, _Rest} <- Tuples],
    Rest = [Rest || {_Row, Rest} <- Tuples],
    split_cols(Rest, [Col|Acc]).

A.4 Fannkuchredux

% The following test is based on an entry in
% The Computer Language Benchmarks Game
% http://benchmarksgame.alioth.debian.org/
%%
%% Contributed by : Alkis Gotovos and Maria Christakis, 13 Nov 2010

-module(fannkuchredux).
-export([[prepare/0, bench/1, main/1]].
prepare() -> {nil, [{MODULE, lists, epiphany, epiphany_server}]}.

bench(nil) ->
    main().

main([Arg]) ->
    main(list_to_integer(Arg)),
    halt(0);
main(N) when N > 0 ->
    {MaxFlips, Checksum} =
    case N of
        1 -> {0, 0};
        _Other ->
            Chunk = fact(N - 1),
            divide(0, N, lists:seq(1, N), Chunk),
            join(N, 0, 0)
    end,
    io:format("-p-nPfannkuchen(-p)_{n}=-p-n", [Checksum, N, MaxFlips]),
    {MaxFlips, Checksum}.

divide(N, N, _L, _C) -> ok;
divide(N, MaxN, [H|T] = List, Chunk) ->
    Self = self(),
    {Spawn, _Count} = colib:spawn_and_count(),
    Fun = fun() ->
        work(N, List, N * Chunk, (N + 1) * Chunk, MaxN, 0, 0, Self)
    end,
    Spawn(Fun),
    divide(N + 1, MaxN, T ++ [H], Chunk).

join(0, MaxFlips, Checksum) -> {MaxFlips, Checksum};
join(N, MaxFlips, Checksum) ->
    receive
        {Flips, Sum} -> join(N - 1, max(MaxFlips, Flips),
            Checksum + Sum)
    end.

work(_P, _L, Index, Index, _R, MaxFlips, Checksum, Target) ->
    Target ! {MaxFlips, Checksum};
work(Proc, List, Index, MaxIndex, R, MaxFlips, Checksum, Target) ->
    reset(R),
    {Flips, Sum} = flip_sum(Index, List),
    NewFlips = max(Flips, MaxFlips),
    NewSum = Checksum + Sum,
    {NewList, NewR} = next(Proc, List, 1),
    work(Proc, NewList, Index + 1, MaxIndex, NewR, NewFlips,
NewSum, Target).

next(Proc, List, R) ->
   NewList = next_aux(R, List),
   case put(R, get(R) - 1) of
      1 -> next(Proc, NewList, R + 1);
      _Other -> {NewList, R}
   end.

next_aux(1, [E1, E2|T]) -> [E2, E1|T];
next_aux(2, [E1, E2, E3|T]) -> [E2, E3, E1|T];
next_aux(3, [E1, E2, E3, E4|T]) -> [E2, E3, E4, E1|T];
next_aux(R, [H|T]) ->
   \{Front, Back\} = lists:split(R, T),
   Front ++ [H] ++ Back.

flip_sum(Index, List) ->
   Flips = flip(List, 0),
   Sum =
   case Index band 1 of
      0 -> Flips;
      1 -> -Flips
   end,
   \{Flips, Sum\}.

flip([1|_T], N) ->
   N;
flip([2, E1|T], N) ->
   flip(E1, 2|T, N + 1);
flip([3, E1, E2|T], N) ->
   flip([E2, E1, 3|T], N + 1);
flip([4, E1, E2, E3|T], N) ->
   flip([E3, E2, E1, 4|T], N + 1);
flip([5, E1, E2, E3, E4|T], N) ->
   flip([E4, E3, E2, E1, 5|T], N + 1);
flip([6, E1, E2, E3, E4, E5|T], N) ->
   flip([E5, E4, E3, E2, E1, 6|T], N + 1);
flip([7, E1, E2, E3, E4, E5, E6|T], N) ->
   flip([E6, E5, E4, E3, E2, E1, 7|T], N + 1);
flip([8, E1, E2, E3, E4, E5, E6, E7|T], N) ->
   flip([E7, E6, E5, E4, E3, E2, E1, 8|T], N + 1);
flip([9, E1, E2, E3, E4, E5, E6, E7, E8|T], N) ->
   flip([E8, E7, E6, E5, E4, E3, E2, E1, 9|T], N + 1);
flip([10, E1, E2, E3, E4, E5, E6, E7, E8, E9|T], N) ->
   flip([E9, E8, E7, E6, E5, E4, E3, E2, E1, 10|T], N + 1);
flip([11, E1, E2, E3, E4, E5, E6, E7, E8, E9, E10|T], N) ->
   flip([E10, E9, E8, E7, E6, E5, E4, E3, E2, E1, 11|T], N + 1);
flip([12, E1, E2, E3, E4, E5, E6, E7, E8, E9, E10, E11|T], N) ->
   flip([E11, E10, E9, E8, E7, E6, E5, E4, E3, E2, E1, 12|T], N + 1);
flip([H|T] = List, N) ->
    {First, Last} = lists:split(H, List),
    flip(lists:reverse(First) ++ Last, N + 1).

reset(1) -> ok;
reset(N) -> put(N - 1, N), reset(N - 1).

fact(1) -> 1;
fact(N) -> N * fact(N - 1).

A.5 Utility and Harness Scripts

A.5.1 Driver
The driver module was the benchmarking harness.
-module(driver).

-export([run/1, scaling/1]).

?-define(SAMPLES, 10).
?-define(US_PER_S, 1000000).

?-define(STATS_FORMAT,
    "|t-p\t-p\t-p\t-p\t-p\t-p\t-p\t-p\t-p\t-p\t-p|").
?-define(STATS_HEADERS, [time, time_sd, reds, reds_sd,
    cycles, cycles_sd, fetch_stall, fs_sd, load_stall, ls_sd]).
?-define(STATS_PATTERN, {Time, TimeSD, Reds, RedsSD, Cycles,
    CyclesSD, Zero, ZeroSD, One, OneSD}).
?-define(STATS_FMTARGS, [Time, TimeSD, Reds, RedsSD, Cycles,
    CyclesSD, Zero, ZeroSD, One, OneSD]).

run([Outletfile|Benchmark]) ->
    case epiphany:state() of
        booting ->
            timer:sleep(100),
            run([Outletfile|Benchmark]);
        _ ->
            ok, IoDev = file:open(atom_to_list(Outletfile), [write]),
            io:format(IoDev, "-p" ?STATS_FORMAT \n",
            [benchmark | ?STATS_HEADERS]),
            io:format("-p" ?STATS_FORMAT "\n",
            [benchmark | ?STATS_HEADERS]),
            try run(IoDev, Benchmarks)
            after
                ok = file:close(IoDev),
                erlang:halt()
        end.

57
run(_IoDev, []) -> ok;
run(IoDev, Benchmark) when is_atom(Benchmark) ->
  Pregen = prepare(Benchmark),
  ?STATS_PATTERN = bench(Benchmark, [Pregen], ?SAMPLES),
  io:format(IoDev, "-p" ?STATS_FORMAT "\n",
              [Benchmark | ?STATS_FMTARGS]),
  io:format("-p" ?STATS_FORMAT "\n",
              [Benchmark | ?STATS_FMTARGS]),
  ok;
run(IoDev, [Benchmark|Benchmark]) ->
  run(IoDev, Benchmark),
  run(IoDev, Benchmarks).
scaling([Outfile|Benchmark]) ->
case epiphany:state() of
  booting ->
    timer:sleep(100),
    scaling([Outfile|Benchmark]);
  _ ->
  {ok, IoDev}
    = file:open(atom_to_list(Outfile), [write]),
    io:format(IoDev, "-p\t-p" ?STATS_FORMAT "\n",
                  [benchmark, cores | ?STATS_HEADERS]),
    io:format("-p\t-p" ?STATS_FORMAT "\n",
                  [benchmark, cores | ?STATS_HEADERS]),
    try scaling(IOdev, Benchmarks)
    after
    ok = file:close(IOdev),
    erlang:halt()
  end.
end.
scaling(_IoDev, []) -> ok;
scaling(IoDev, [Benchmark|Benchmark]) ->
  {_Spawn, Count} = colib:spawn_and_count(),
  Pregen = prepare(Benchmark),
  scaling(IoDev, Benchmark, Pregen, Count),
  scaling(IoDev, Benchmarks).
scaling(_IoDev, _Benchmark, _Pregen, 0) -> ok;
scaling(IoDev, Benchmark, Pregen, Cores) ->
  ?STATS_PATTERN = bench(Benchmark, [Pregen, Cores], ?SAMPLES),
  io:format(IoDev, "-p\t-p" ?STATS_FORMAT "\n",
            [Benchmark, Cores | ?STATS_FMTARGS]),
  io:format("-p\t-p" ?STATS_FORMAT "\n",
            [Benchmark, Cores | ?STATS_FMTARGS]),
  scaling(IoDev, Benchmark, Pregen, Cores div 2).
prepare(Benchmark) ->
  {Pregen, Modules0} = Benchmark:prepare(),
Modules = [Benchmark, epiphany, epiphany_server | Modules0],
Spawn =
case epiphany:state() of
    offline ->
        lists:foreach(fun code:ensure_loaded/1, Modules),
        fun erlang:spawn/1;
    online ->
        lists:foreach(fun code:ensure_loaded_epiphany/1, Modules),
        fun epiphany:spawn/1
end,
%% Roundtrip to ensure everything is ready before we start
%% measuring
Self = self(),
Spawn(fun() -> Self ! ready end),
receive ready -> ok after 10000 -> error(timeout) end,
PreGen.

bench(Benchmark, Args, Samples) ->
    {Times, Reds, Cycles, FetchStalls, LoadStalls} = bench(Benchmark, Args, [], [], [], [], Samples),
    {avg(Times) / ?US_PER_S, avg(Args),
     stdev(Times) / ?US_PER_S, stdev(Args),
     avg(Cycles), avg(Args),
     stdev(Cycles), stdev(Args),
     case FetchStalls of nan -> nan; _ -> avg(FetchStalls) end,
     case FetchStalls of nan -> nan; _ -> stdev(FetchStalls) end,
     case LoadStalls of nan -> nan; _ -> avg(LoadStalls) end,
     case LoadStalls of nan -> nan; _ -> stdev(LoadStalls) end}.

    avg([]) -> nan;
    avg(List) -> lists:sum(List) / length(List).
    stdev([]) -> nan;
    stdev([_]) -> nan;
    stdev(List) ->
        Len = length(List),
        Mean = lists:sum(List) / Len,
        math:sqrt(lists:sum([(X-Mean)*(X-Mean) || X<-List])
                        / (Len - 1)).

bench(_Benchmark, _Args, Times, Reds, Cycles, FetchStalls, LoadStalls, 0) ->
    {Times, Reds, Cycles, FetchStalls, LoadStalls};
bench(Benchmark, Args, Times, Reds, Cycles, FetchStalls0, LoadStalls0, Samples) when Samples > 0 ->
    case Samples rem 2 of
        1 -> lib:select_timers(clk, ext_fetch_stalls);
0 -> colib:select_timers(clk, ext_load_stalls)
end,
timer:sleep(50),
_ = colib:poll_stats(),
\{Time, _\} = timer:tc(Benchmark, bench, Args),
\{Zero, One, Red\} = colib:poll_stats(),
Quotient = case Zero of 0 -> nan; _ -> One / Zero end,
io:format("-p_samples_of\(\_\)p_remo\(\_\)remaining:\(\_\)p_s,\(\_\)reds:\(\_\)p_T0:\(\_\)
-p,\(\_\)T1:="p\n",
\{Samples - 1, 
  Benchmark, Time / \(?US_PER_S, 
  Red, Zero, One\}\},
\{FetchStalls, LoadStalls\} =
case Samples rem 2 of
  _ when Quotient =:= nan -> \{nan, nan\};
  1 -> \{[Quotient | FetchStalls0], LoadStalls0\};
  0 -> \{FetchStalls0, [Quotient | LoadStalls0]\}
end,
bench(Benchmark, Args, \{Time | Times\}, \{Red | Reds\},
  \{Zero | Cycles\}, FetchStalls, LoadStalls, Samples - 1).

A.5.2 Colib
The colib module provided reusable concurrency abstractions, as well as wrappers
for the spawn functions that collects the desired statistics.
-module(colib).
-export([pmap/2, pmap/3, \% pforeach/2,
  spawn_and_count/0, 
  poll_stats/0, select_timers/2]).
pmap(Fun, List) -> 
    Collector = self(),
    Indices = lists:seq(0, length(List)-1),
    Jobs = lists:zip(Indices, List),
    Work = fun({Ix, El}) -> Collector ! {Ix, Fun(El)} end,
    spawn_link(Fun() -> pforeach(Work, Jobs) end),
    [receive {Ix, El} -> El end || Ix <- Indices].
pmap(M, F), [], List) -> pmap(fun M:F/1, List); 
pmap(M, F), [A], List) -> pmap(fun (E) -> M:F(E, A) end, List); 
pmap(M, F), Extra, List) -> 
    pmap(fun(E) -> erlang:apply(M, F, [E|Extra]) end, List); 
pmap(Fun, [], List) -> pmap(Fun, List); 
pmap(Fun, [A], List) -> pmap(fun (E) -> Fun(E, A) end, List); 
pmap(Fun, Extra, List) -> 
    pmap(fun(E) -> erlang:apply(Fun, [E|Extra]) end, List).

\% Does not wait for completion
pforeach(Fun, Jobs) ->
    Self = self(),
    
    {Spawn, Count} = spawn_and_count(),
    Workers = [Spawn(fun()) -> worker(Fun, Self) end]
    || _ <- lists:seq(1, Count)],
    workserver(Workers, Jobs).

workserver(Workers, [Job|Jobs]) ->
    receive
        {get_job, Worker} ->
            Worker ! {job, Job},
            workserver(Workers, Jobs)
    end;
    workserver(Workers, []) ->
        lists:foreach(fun(W) -> W ! stop end, Workers).

worker(Fun, Jobserver) ->
    Jobserver ! {get_job, self()},
    receive
        stop -> ok;
        {job, Job} ->
            Fun(Job),
            worker(Fun, Jobserver)
    end.

spawn_and_count() ->
    case try epiphany:state() catch error:undef -> offline end of
    offline ->
        {fun regular_spawn_link/1,
         erlang:system_info(schedulers_online)};
    booting ->
        timer:sleep(100),
        spawn_and_count();
    _ ->
        start_server(),
        {fun epiphany_spawn_link/1, epiphany:count()}
    end.

start_server() ->
    case whereis(colib_server) of
    undefined ->
        {Self, ReportRef} = {self(), make_ref()},
        {Pid, MonRef} =
            spawn_monitor(
                fun() -> server_entry(Self, ReportRef) end),
        receive
            ReportRef ->
                demonitor(MonRef, [flush]),
                Pid;
            
            {'DOWN', MonRef, _, _, Reason} ->
                case whereis(colib_server) of


61
```erlang
undefined -> error(Reason);
Pid2 -> Pid2
end;

Pid -> Pid
end.

-record(state, {zero_sum = 0, one_sum = 0, red_sum = 0, processes = []}).

server_entry(Report, Ref) ->
    register(colib_server, self()),
    Report ! Ref,
    server_loop(#state{}).

server_loop(State0 = #state{processes = Processes}) ->
    receive
        {spawned, Pid} ->
            monitor(process, Pid),
            server_loop(State0#state{processes = [Pid|Processes]});
        {died, Pid, Stats} ->
            server_loop(server_handle_died(Pid, Stats, State0));
        {’DOWN’, _, process, Pid, Abnormal}
            when Abnormal /= normal ->
                server_loop(State0#state{processes = lists:delete(Pid, Processes)});
        {poll, Asker, Ref} ->
            State = #state{zero_sum=ZeroSum, one_sum=OneSum, red_sum=RedSum} = server_poll_loop(State0),
            Asker ! {Ref, {ZeroSum, OneSum, RedSum}},
            server_loop(State#state{zero_sum = 0, one_sum = 0, red_sum = 0});
        _Other ->
            server_loop(State0)
    end.

server_poll_loop(State = #state{processes = []}) -> State;
server_poll_loop(State = #state{processes = Processes}) ->
    receive
        {died, Pid, Stats} ->
            server_poll_loop(server_handle_died(Pid, Stats, State));
        {’DOWN’, _, process, Pid, Abnormal}
            when Abnormal /= normal ->
                server_poll_loop(State#state{processes = lists:delete(Pid, Processes)});
    end.
```

62
server_handle_died(Pid, {Zero, One, Reds},
    State = #state{zero_sum = ZeroSum,
                  one_sum = OneSum,
                  red_sum = RedSum,
                  processes = Processes}) ->
    State#state{zero_sum = ZeroSum + Zero,
                one_sum = OneSum + One,
                red_sum = RedSum + Reds,
                processes = lists:delete(Pid, Processes)}.

-spec poll_stats() -> {integer(), integer(), integer()}.
poll_stats() ->
    start_server() ! {poll, self(), Ref = make_ref()},
    receive {Ref, Answer} -> Answer end.

select_timers(ZeroTimer, OneTimer) ->
    application:set_env(colib, timers, {ZeroTimer, OneTimer}).

regular_spawn_link(Fun) ->
    spawn_link(
        fun() ->
            colib_server ! {spawned, self()},
            try Fun()
            after
                {reductions, Reds} = process_info(self(), reductions),
                colib_server ! {died, self(), {0, 0, Reds}}
            end
        end).

epiphany_spawn_link(Fun) ->
    epiphany:spawn_link(
        fun() ->
            colib_server ! {spawned, self()},
            {ZeroConf, OneConf} = application:get_env(colib, timers, {off, off}),
            _ = epiphany:timers(ZeroConf, OneConf),
            try Fun()
            after
                {Zero, One} = epiphany:timers(off, off),
                {reductions, Reds} = process_info(self(), reductions),
                colib_server ! {died, self(), {Zero, One, Reds}}
            end
        end).

63