2024 22nd ACM-IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE)

Higher-order Hardware: Implementation and Evaluation of the Cephalopode Graph Reduction Processor

Jeremy Pope
Department of CSE
Chalmers University of Technology
Gothenburg, Sweden
popje@chalmers.se

Carl-Johan H. Seger
Department of CSE
Chalmers University of Technology
Gothenburg, Sweden
secarl@chalmers.se

Henrik Valter
Department of CSE
Chalmers University of Technology
Gothenburg, Sweden
valterh@chalmers.se

***Abstract*—A major challenge with the practical deployment of Internet-of-Things (IoTs) is how to develop the high-quality code needed in order to produce robust and secure IoT devices. In other domains, high-level programming languages have shown to be efficient vehicles towards this. However, the very limited compute power provided by IoT devices have made it difficult to apply the same approach to IoT devices. The Cephalopode processor is an attempt at implementing a low power hardware device directly aimed at running a high-level functional language. By integrating many resource-heavy tasks like garbage collection and arbitrary precision arithmetic into dedicated hardware, the Cephalopode processor explores the hypothesis that high-level functional languages can be used even for low-power IoT devices.**

This paper presents the implementation and evaluation of the Cephalopode processor. We discuss the approach taken, the compiler and the architecture of the processor. We also describe the design process and design considerations.

After implementation and synthesis we compare the processor to a conventional RISC-V processor running a functional language software environment. We also compare Cephalopode with running handwritten C code on the RISC-V processor.

I. Introduction

The Internet of Things (IoT) conceives a future where “things” will be interconnected by means of suitable information and communication technologies. This connectivity provides many opportunities, but also vulnerabilities.

IoT devices, like all embedded systems, tend to be designed with tight resource constraints. Energy consumption in particular needs to be kept at a minimum, since these devices usually are battery-powered and are expected to run for several years.

Almost exclusively, IoT devices are programmed with low-level languages such as C and C++ due to their minimal runtime systems and low-level hardware capabilities. This simplicity comes at the cost of the abstractions that higher-level languages provide, especially memory safety. In fact, there are already reports on IoT security being breached, such as smart fridges being hacked to reveal Gmail passwords [1], cars being remotely controlled [2], and home devices being hijacked to launch massive DDoS attacks [3].

With these examples in mind, it becomes clear that we need safe, high-level languages for programming IoT. Functional programming languages such as Haskell provide an appealing alternative [4]. They have been proven to be quite good at providing programming environments in which security and privacy guarantees can be provided, without imposing demanding user constraints [5]. Strong typing and type inference help in catching many errors at compile time. They also lend themselves well to formal verification, which is not only invaluable for critical systems where correctness and safety is crucial, but especially where errors are difficult to correct. Consider for example an IoT device that executes a program stored in read-only memory, where said program is later found to have critical flaws. In the best case, the problem is quickly found and mitigated, likely at a large cost. In the worst case, the problem is not found until damage—or worse—occurs because of the flaw.

Unfortunately, the high level of abstraction comes at a cost. The Haskell runtime system is quite substantial, and has proven very difficult to run in resource-constrained environments. Supporting the entire runtime system in tiny embedded devices that are expected to run for several years on small batteries poses significant engineering challenges. In addition, the runtime system requires garbage collection, making it difficult to provide the necessary performance guarantees for systems with real-time system requirements, such as many IoT systems.

An alternative to an extensive software runtime system would be to create a dedicated hardware design aimed at executing the high-level language natively while providing the various services needed directly. This paper presents the Cephalopode processor, which is a first realization of this vision.

The Cephalopode processor is based on combinator graph reduction, a simple execution mechanism for lazy functional languages. The fundamental concept is that the high-level

This work was partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation and by a grant from the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023).

978-1-7281-9148-5/20/$31.00 © 2020 IEEE

language is compiled into a computational graph consisting of a small set of simple combinators. The program execution would then entail to perform a step-by-step reduction of this computational graph. An analogy is how high-level imperative languages are compiled into a finite set of opcodes that are then executed one-by-one. We direct the reader to [6] for an introduction to the subject of graph reduction.

There is one class of bugs in C/C++ based programs that is particular common in IoT devices: arithmetic overflow. Since IoT processors often are very simple, and thus in many cases do not contain hardware support for floating point numbers, various sized integers—typically 8, 16, or 32 bit ones, signed or unsigned—are often used to represent values. Unfortunately, this means that a common failure is arithmetic overflow. If we add to this the common technique of using scaled integer arithmetic to improve accuracy, it is very easy to write code that for some particular boundary conditions will cause a (silent) arithmetic overflow, causing unpredictable behavior. To drastically reduce this possibility, the built-in arithmetic in Cephalopode uses a form of arbitrary precision, in which integers grow and shrink in size as needed. Doing this in software is virtually impossible without imposing serious performance degradation (which we show in the results section). For hardware, this ability comes with a rather modest cost and appears to be well worth having.

Finally, since an IoT device is often used with hard realtime constraints, Cephalopode's graph reduction engine is tightly integrated with a true concurrent garbage collector, providing much needed deterministic performance guarantees.

As will be explained further in Section II, there been several historical attempts, especially from the 1980s and 1990s, at creating hardware intended for functional language execution [7]–[11]. Unfortunately, these have always been outperformed by software implementations on conventional machines, if not immediately then within a few process generations. This was often due to high memory traffic, which has become increasingly expensive in high-performance processors. The landscape of computer architecture has however changed since those times. Now, with the end of Moore's Law and Dennard Scaling, custom computer architectures such as Cephalopode can now be competitive with software implementations on conventional architectures when targeting resource-constrained environments. Energy consumption especially is an area where we believe custom architectures can be especially potent, which, as previously mentioned, is central in IoT. Furthermore, in the this domain where processor frequencies are in the tens of MHz, memory traffic is not as expensive as in high-performance computers. This makes architectures with higher memory traffic more feasible than in the high-performance domain.

A. Our contribution

To summarize, we present the following contributions.

- We evaluate the conjecture of whether designing dedicated hardware for functional language execution is a feasible approach for resource-constrained systems, especially in terms of energy.
- We provide a detailed description of the design of one such hardware implementation as well as suggestions to how it could be improved further.
- In an apples-to-apples comparison, we provide measurements showing that the resulting post-synthesis processor has 1-2 orders of magnitude lower energy consumption than a reference RISC-V machine running combinator graph reduction in software and about twice the energy consumption running equivalent code written in C on the RISC-V.
- We also provide measurements summarizing the cost of using arbitrary precision arithmetic and functional language execution by comparing our processor against "bare bones" C code.

B. Paper structure

The paper is structured as follows. In Section II, we present related work on functional language processors. In Section III, we present a detailed description of the Cephalopode architecture. Section V describes the Cephalopode compiler. In Section VI, we describe the methodology and tool set for synthesizing and evaluating the processor. We also describe the reference RISC-V core running code emitted by MicroHs that Cephalopode is compared against. In Section VII, we present and evaluate the results. Finally in Section VIII we describe our plans for further development of Cephalopode.

II. Related work

The use of combinators to evaluate functional programs was pioneered by the late David Turner in his 1979 paper [12], which presented a software implementation but hinted at the possibility of a hardware one. Several hardware and abstract machines designed to perform or accelerate combinator graph reduction emerged thereafter.

SKIM [13] and NORMA [14] implement combinator-based graph reduction in hardware, albeit microcoded.

Hughes [15] presents a method of compilation that uses "super-combinators" derived from the source program rather than a fixed set, improving program size and speed. They were later hardware-accelerated in TIM [16].

Augustsson and Johnsson described the G-machine [17], an abstract machine suitable for running functional programs but amenable to efficient implementation on a standard computer processor.

The abstract machine TIGRE [18] provides a faster approach to combinator graph reduction, with the potential to be extended to super-combinators.

The Spineless Tagless G-machine [19], a descendant of the G-machine, serves as the basis of Haskell compilation in the Glasgow Haskell Compiler (GHC [20]) to this day. GHC remains the state-of-the-art Haskell compiler to this day.

Several recent works also explore the evaluation of functional programs. The Reduceron [21] aims to exploit parallelism to rapidly evaluate functional programs on FPGAs, with an explicit goal of trying to perform as many reduction steps per clock cycle as possible.

Accetti et al. introduce Lambda-One [22], an architecture for evaluating functional programs on FPGAs using combinators. Although compelling in several respects, it is unclear how the stop-the-world garbage collection system fares with data that is promoted into the external memory; it seems likely that such an algorithm would introduce substantial latency given the clock frequency of the processor (100MHz) and size of the DRAM (512MB). Energy consumption and performance relative to traditional architectures are also not clear.

III. Design

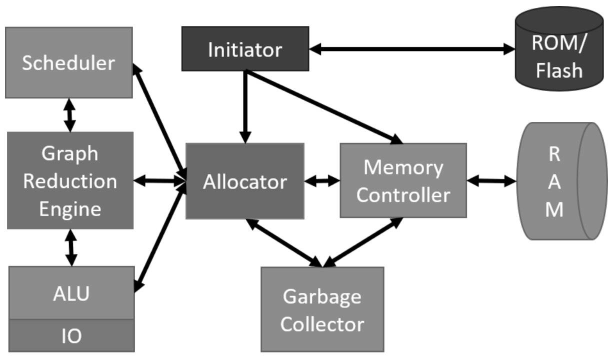

Fig. 1. Cephalopode architecture. I/O primitives are not yet implemented.

Cephalopode is comprised of several units (shown in Fig. 1) that work together to run functional programs. The *graph reduction engine* is the core of the processor, responsible for traversing the graph and performing computation by carrying out reduction steps. Inside the reduction engine are the *primitive unit*, which performs arithmetic using the *ALU*, and the *combinator unit*, which performs combinator reductions. The *memory controller* manages access to main memory, with additional behavior to be described in later subsections. The *allocator* provides new nodes to the reduction unit as needed, and conversely the *garbage collector* recycles nodes that are no longer in use. The *initiator* loads the program from ROM into RAM at boot, and the *scheduler* manages the high-level operation of the processor. This section will first describe the representation used for programs, and then discuss the details of how they are run in relation to the units listed above.

A. Graph model

Programs—in essence lambda expressions—are stored as graphs in Cephalopode's main memory. Graph nodes are a fixed size, allowing memory to be designed so that one graph node corresponds to one word. Each node stores a tag that indicates what type of node they are. Node types include `APP`, representing function application, `COMB` for combinators, `PRIM` for primitive functions (e.g. arithmetic), `INT` and `AINT` for multiple-precision integer data, `CONS` and `NIL` for lists, `INDIR` for "indirections" to other nodes, and `FREE` for nodes in the free list.

The remaining fields vary based on the node type. `APP` and `CONS` nodes have two address fields `left` and `right`; for the former these are the function and argument, and for latter the "head" and "tail" pointers. `APP` nodes also contain an address field `up`, used in lieu of a stack in order to backtrack toward the root of the graph during reduction. `COMB` and `PRIM` nodes contain a field that indicates which combinator or primitive function they refer to. `INT` nodes contain only integer data, for integers that are small enough to fit in one node, while `AINT` nodes store some integer data but also the address of a subsequent `AINT` node in order to allow the representation of longer integers. `NIL` nodes store no additional data. `INDIR` and `FREE` nodes each contain a pointer, for `INDIR` to the referenced node, and for `FREE` to the next node in the free list.

B. Reduction algorithm

Reduction on a graph is carried out by beginning at the graph's root node, and traversing the `left` pointer down the spine of the graph so long as application (`APP`) or indirection (`INDIR`) nodes are found. The `up` pointer of each `APP` node is modified to point to the previous `APP` node in the spine. (This is similar to pointer reversal in some functional programming architectures, which re-uses the `left` pointer for this task. By using a dedicated pointer the size of each node is increased but there is no need to rectify pointers while backtracking, avoiding a substantial number of memory writes.) When an indirection node is encountered, its pointer is followed immediately and the node is otherwise ignored; in particular `up`-pointers always skip over them to an `APP` node instead.

When a non-application/indirection node is encountered, a reduction step may take place. For combinators and primitive functions, the following takes place:

1) The arity N of the combinator/primitive is determined from a look-up table.
2) N arguments are gathered by backtracking up the spine via the `up` pointers, and recording the `right` pointers seen along the way.
3) The address of the *redex*, the root of the sub-expression being reduced, is recorded. (It is the final application node encountered during the backtracking.)
4) The computation indicated by the combinator/primitive is performed.
5) The redex node is overwritten with the computation result.
6) The pointer used for traversal is updated.

Traversal may then continue, and the process repeats. A naïve approach would continue traversal from the redex, but this is not optimal: combinators such as `S` add application nodes as left children of the redex, which will certainly need to be traversed. So rather than always resuming from the redex, traversal resumes from the furthest node that is known to be in the spine based on the combinator that was applied.

C. Multiple-precision arithmetic

The arithmetic used in Cephalopode is *multiple-precision*: integers may grow beyond a single node (`INT`) to a linked list of nodes (AINT). To reduce memory traffic, the arithmetic unit uses a 128-word internal register file to buffer operands and intermediate results. Currently this limits the size of valid arithmetic operations, making the arithmetic not fully arbitrary-precision. This is not fundamental to the architecture, however: the arithmetic unit could equally well spill into main memory when operands are too large to use only the internal register file. Care is taken to ensure that arithmetic on small integers—the most common case—remains fast.

D. Indirection nodes

For projections such as the `K` combinator, which produce an already-existing graph node as a result, simply overwriting the redex with the result can lead to duplicated work later on. For example, reducing `(K (f x) y)` results in `(f x)`, but overwriting the redex with this application would result in two application nodes for `(f x)`: the original one in the argument (which may still be referenced elsewhere due to sharing), and the new one written to the redex. Being separate application nodes despite referencing the same function and same argument, evaluating one of these would not bear fruit for the other, leading to redundant effort if `f` were computationally expensive.

To avoid this, the redex is instead overwritten with an indirection (INDIR) node with a pointer to the already-existing result. Since indirections are followed automatically during graph reduction, this effectively unifies the redex with the result, and no duplication of work occurs.

E. Strictness

Primitive functions, such as addition, are strict: they require that their operands are already fully evaluated. In order to avoid having to recursively evaluate the arguments (which incurs additional complexity and requires either explicitly or implicitly maintaining a second stack), a clever trick from the Reduceron [21] is used, which we refer to here as the *swap rule*: Whenever an application `(v f)` is found, and `v` is a non-function value (e.g. integer), it is reduced to `(f v)`. This can be used as follows: in order to evaluate a strict unary function `f` applied to an expression `e`, rather than beginning with `(f e)`, we begin with `(e f)`. Reduction will make its way down the left side of the application, eventually reducing `e` to some value `v`, leaving us with `(v f)`, and then the above rule will swap the application to `(f v)`, and evaluation can continue as originally intended. This transformation (beginning with `(e f)` instead of `(f e)`) can be generalized to higher arities to the same effect, while only using the single swap rule described above. Provided that functions are never strict about arguments of function type (such as combinators or primitives), this allows strict functions to be implemented without a significant increase in hardware complexity.

F. Garbage collection

As graph reduction requires frequent allocation of new nodes, and it is difficult to predict when nodes will no longer be in use, the nodes that are not currently in use are organized in a linked list called the *free list*, from which allocation draws. In order to return nodes that are no longer in use to this list, tracing garbage collection is used; it is noted that reference counting is insufficient since cycles may be present due to (mutually) recursive computations.

However, the use of a tracing garbage collector in an IoT context presents some difficulty: stopping the process long enough to perform the collection could introduce significant delays, interfering with operation. To avoid this, Cephalopode carries out garbage collection largely in parallel with the running process. Furthermore, the garbage collection can be carried out in periods when the process is not actively using main memory (e.g. during arbitrary-precision arithmetic, which uses a local cache), minimizing contention for memory access.

Simply running a tracing garbage collection algorithm in parallel to a computation will not work by itself, since changes to the program graph while the garbage collector is in the process of exploring it may cause the latter to miss pointers to live memory objects. However, Yuasa [23] observes that since garbage always remains garbage until it is freed, it is safe for a tracing garbage collection algorithm to operate on a *snapshot* of a graph, even while the graph continues to evolve after the snapshot is taken. Any garbage identified in the snapshot must still be garbage once the marking phase of the collector is finished, and is thus safe to reclaim regardless of how the graph has changed. The only consequence of the discrepancy is that new garbage will be missed and thus not reclaimed, leaving the garbage collector one garbage collection cycle behind.

The approach used by Yuasa does not use a literal snapshot, instead it uses write-barriers to evacuate pointers that would be overwritten, placing them into the garbage collector's stack to ensure that they will be explored. This makes write operations very slow, however, as each write requires a read to be carried out first. Cephalopode instead opts to make writes comparatively efficient by redirecting writes to alternative memory locations, leaving the original data intact for the garbage collector to explore. This requires twice as much memory as may be in active use, similar to a copying garbage collector [24], albeit also with a stack. By doing this, an exact snapshot of the graph at the beginning of garbage collection is maintained, and basic correctness of the garbage collection process follows as a result. Furthermore, as will be described in detail later, the snapshot can be taken nearly instantaneously, avoiding causing delays to the graph reduction when garbage collection begins.

Cephalopode uses a simple mark-and-sweep algorithm for the garbage collection itself, with additional operations to pause graph reduction, manage the snapshot, and atomically modify the free list. The set of roots for garbage collection consists of the root pointer of the program graph, and the head of the free list: provided that the snapshot is taken between reduction steps, all data we do not wish to reclaim (live data and the free list) can be reached from these. As a consequence there is no need to gather roots from a call stack, as would be needed with a traditional system, minimizing delays.

Pseudocode for the garbage collector is given in Listing 1, with some minor simplifications. Graph reduction is only stopped while the snapshot is taken and the two roots are marked and pushed onto the garbage collection stack—in other words for a brief and constant-bounded time.

```
function main():
    loop:
        wait(allocated_nodes > threshold)
        wait(take_snapshot_ready)
        // Preparation and root-finding
        pause_graph_reduction()
        take_snapshot()
        mark_and_push(freelist_head)
        mark_and_push(graph_root)
        resume_graph_reduction()
        // Hard work
        mark_phase()
        finished_snapshot()
        sweep_phase()

function mark_phase():
    while stack not empty:
        ptr = pop()
        n = read_snapshot(ptr)
        if n.type in [CONS, APP]:
            mark_and_push(n.right)
            mark_and_push(n.left)
        else if n.type in [FREE, INDIR, AINT]:
            mark_and_push(n.left)

function sweep_phase():
    // Construct linked list of reclaimed nodes
    reclaimed_head = null
    reclaimed_tail = null
    for sweeper in HEAP_MIN...HEAP_MAX:
        if get_mark(sweeper) == 1:
            set_mark(sweeper, 0)
            compress_indirections(sweeper)
        else:
            m = node()
            m.type = FREE
            m.left = reclaimed_head
            write_mem(sweeper, m)
            reclaimed_head = sweeper
            if reclaimed_tail == NULL:
                reclaimed_tail = sweeper
    // Atomically prepend to free list
    if reclaimed_tail != null:
        n = read_mem(reclaimed_tail)
        f = checkout_freelist_head()
        n.left = f
        write_mem(reclaimed_tail, n)
        checkin_freelist_head(reclaimed_head)

function mark_and_push(p):
    if p != NULL and get_mark(p) == 0:
        set_mark(p, 1)
        push(p)
```

Listing 1. Cephalopode's garbage collection algorithm.

The mark bits are packed into 32-bit words and stored in a small, high-speed memory located in the garbage collector. A one word cache is used to avoid performing a read-modify-write when setting a single mark bit; the access patterns of the garbage collector (checking a mark bit before modifying it) ensure that when a mark bit is changed the mark word it belongs to is already in the cache, and can be used without an extra read operation.

The stack is also stored in a separate memory. During marking, the garbage collector always pops from the stack after no more than two mark-and-push sequences. Thus if the graph memory can contain a maximum of N nodes, the depth of the stack will be at maximum $\lfloor N/2 \rfloor + 1$. The stack is sized based on this worst-case scenario.

G. Indirection chain compression

Unfortunately, in graph reduction systems that use indirection nodes, ordinary garbage collection is not quite sufficient: space can be consumed by growing chains of indirection nodes. In a stop-the-world garbage collector these can be removed fairly easily, but in Cephalopode's case the graph reduction continues even during garbage collection, the usual approach will result in race conditions.

However, it is observed that once a node is an indirection node, it will never change until it is freed: indirection nodes are not treated as part of the spine—just traversed silently—so they are never overwritten during reduction. So if a indirection node X points through a chain of further indirection nodes to a non-indirection node Y, it is safe to modify X to point directly to Y—the only change seen by the reduction engine is that while traversing the graph it may reach Y more quickly than if X had not been modified. Furthermore, this can be carried out on every node in an indirection chain: they can all be modified to point to the node that follows the chain, analogous to path compression in set-merging algorithms [25]. Applying this reduces the indirection chain to separate indirection nodes all pointing at the same destination, thereby reducing all chains to length one. Subsequent garbage collection cycles can then remove any of the indirection nodes that are no longer referenced.

The proliferation of single indirection nodes is not solved by this, and Cephalopode does not as yet make an effort to remove them. However, it is noted that since nodes have at most two child pointers, after chains are compressed indirection nodes may constitute at most two thirds of all live nodes, as opposed to an unlimited proportion of the population.

A convenient place to perform this compression is in the sweep phase of the garbage collector, since it will look at all live nodes, and has a stack available to remember previous nodes in a chain. Using the garbage collector stack for the latter limits the length of compression to the size of the stack (half the maximum number of nodes), but it appears unlikely that a chain would exceed this length during normal operation, and if one were to it would only require a second garbage collection cycle to be fully compressed. The advantage of using the stack is that that fewer reads from main memory are needed, reducing contention. Pseudocode for the procedure is given in Listing 2.

```
function compress_indirections(sweeper):
    n = read_mem(sweeper)
    if n.type == INDIR:
        p = sweeper
        while n.type == INDIR and stack not full:
            push(p)
            p = n.left
            n = read_mem(p)
        end = p
```

```
while stack not empty:
    p = pop()
    mem_write(p, make_indirection_to(end))
```

Listing 2. Cephalopode's indirection chain compression algorithm.

A subtle consequence arises in relation to graph traversal: the reduction engine could be in the process of traversing the chain when it is compressed. In particular, it could be traversing a node in the chain that happens to become an orphan, and will be reclaimed on the next garbage collection cycle. If that happened before the reduction engine continued traversal (unlikely in a single-process system, but conceivable in a multi-process one) then the latter would end up accessing invalid memory. To prevent this, the reduction engine will never pause for garbage collection to begin when it is traversing an indirection chain, instead it will only pause briefly once it reaches a non-indirection node.

H. Snapshot memory

The snapshot functionality is implemented inside of the memory manager, which exposes both normal read and write operations as well as snapshot management: preparing to take, taking, and reading from one. Aside from preparing to take a snapshot—which is carried out incrementally in the background—none of the operations have significant latency. In order to accomplish this, each virtual address m used by the graph reduction engine corresponds to *two* physical addresses $2m$ and $2m+1$ in main memory; it is noted that these are m followed by a least significant bit (LSB) of our choosing. This redundancy allows the storage of both a snapshot version and an up-to-date version of the data associated with virtual address m. A small, high-speed memory housed inside the memory manager keeps track of which is which: for each virtual address m, it stores the LSB for the up-to-date value (denoted $\textit{new}_m$), and the LSB for the snapshot value (denoted $\textit{old}_m$). These, combined with a global bit *mode*, are then used to implement the desired snapshot functionality:

- When $\textit{mode} = \textit{MODE\_NORMAL}$:
  - A read or write to virtual address m uses $\textit{new}_m$ as the LSB.
  - Sometime before taking a new snapshot, set $\textit{old}_m \leftarrow \textit{new}_m$ for each m.
  - To take a snapshot, set $\textit{mode} \leftarrow \textit{MODE\_SNAPSHOT}$.
- When $\textit{mode} = \textit{MODE\_SNAPSHOT}$:
  - A read on m uses $\textit{new}_m$ as the LSB.
  - The snapshot of m may be read by instead using $\textit{old}_m$ as the LSB.
  - A write to m sets $\textit{new}_m \leftarrow \overline{\textit{old}_m}$, and uses that as the LSB.
  - To finish with the snapshot, set $\textit{mode} \leftarrow \textit{MODE\_NORMAL}$.

Taking a snapshot consists only of toggling a *mode* bit, but the preparation beforehand—setting $\textit{old}_m$ equal to $\textit{new}_m$ for each m—requires a number of operations that is linear with respect to the size of memory. However, these operations only concern the high-speed memory inside the memory manager, not the main memory, and can thus be carried out concurrently without introducing contention. Provided some time between garbage collection cycles is intended, no latency is introduced: the preparation can begin as soon as the previous garbage collection cycle is done with marking, and be finished before the next one should begin, allowing the latter to take a snapshot without delay.

By banking physical memory, reads can be accelerated by reading both physical addresses at once, as well as the *new* and *old* bits, and selecting which data to output based on the latter. This is not possible with writes, which require first reading the *new* and *old* bits in order to know which physical address should be written to. However, with a high-speed memory this is still faster than a read from main memory, and if sufficiently quick this can be carried out in the same clock cycle as the main memory write is signalled, as is the case in Cephalopode. It also avoids the need to mark and push the pointers contained in the old value as a Yuasa-style system would need to; instead the garbage collector can access the old value at its leisure.

In summary, the reduction engine sees a potentially slower memory during snapshot mode and when garbage collection is taking place, but never experiences significant delays. The garbage collector only experiences long delays while waiting for the current reduction step to finish, or if it were to begin a new garbage collection cycle very soon after the previous one (i.e., before the memory controller has time to prepare for a new snapshot).

I. Context switching

As described previously, it is necessary to walk back up the spine of a graph during reduction, typically implemented using a stack. By embedding this stack in application nodes through the `up` pointer, rather than maintaining a separate stack, the state of graph reduction can be encapsulated in just two pointers: a pointer to the node where traversal/reduction will continue, and a pointer to the previous application node in the spine.

Context switches then become trivial, provided the graphs are disjoint: once any pending reduction step completes (or is aborted), the two pointers for the current process are stored so that it can be resumed later, and two new pointers are given to the reduction engine to work on. Aside from waiting for a large arithmetic operation to complete, this can be accomplished in only a handful of clock cycles.

Although the current Cephalopode implementation only maintains one process, the context switching mechanism is in fact already in use: it is how the graph reduction is briefly paused at the beginning of garbage collection.

IV. Implementation environment

Cephalopode was designed using the VossII [26] platform, a design and verification system based on the functional programming language fl. Its symbolic simulation capabilities were used to test error-prone parts of the design, for example within the arithmetic unit. Verilog output from VossII was used for the evaluation described in sections VI and VII.

Many parts of Cephalopode—particularly those with complex behavior such as the reduction engine, garbage collector, and snapshot memory controller—were written primarily in Bifröst, a high-level language that can be compiled to hardware descriptions in fl. As well as simplifying the design process, it allowed the automatic generation of clock-gating circuitry, the latter reducing energy consumption by approximately half for the core itself (i.e., excluding memories).

V. Compiler

The compiler for the Cephalopode processor relies heavily on fl, the language of the VossII [26] system, both to provide the front end of the compiler as well as serve as the implementation language for the compiler. The stages of the compiler are shown in Fig. 2.

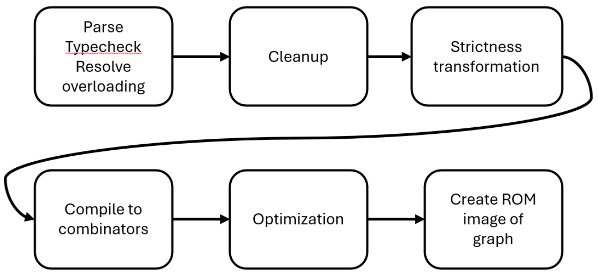

Fig. 2. The stages of the Cephalopode compiler.

First, we rely on fl's internal language interpreter to parse the program, type-check it, and resolve any overloaded functions. The resulting lambda expressions are then obtained through the reflection mechanism of fl.

In the next stage, strict primitives are identified and the graph is transformed to enable call-stack free evaluation. This is largely a traversal over the graphs. However, if a strict function is unsaturated, e.g., because it is partially applied, additional lambda abstractions and variables are added so that all strict functions are fully applied.

Originally, Cephalopode was designed to use a fixed set of combinators similar to the ones described in [6]. However, after inspecting the graphs obtained after the strictness rearrangements required for Cephalopode, we discovered a very common pattern. As a result, we decided that an additional combinator would be useful. However, the number of arguments to this needed combinator varies and thus we needed a family of combinators. We named this family of combinators Ln, where n is a numerical value that denotes the depth of the chain. Since we had plenty of space for such a counter in the node structure, we effectively added an unbounded number of combinators. We also added a similar Cn combinator, although it is of more dubious value in the examples we have considered. The reduction rule for all the combinators we use are given in Table I and in Table II we provide some statistics on how effective these new combinators are in reducing the size of the graph to reduce. We show three statistics: 1) how many combinators the program used, 2) the size of the program graph, and 3) the total number of reductions needed to evaluate the program. For our set of benchmarks, the additional combinators have a major impact, reducing the program size by 30% and the number of reductions, and thus directly the run-time of the program, by 31%.

TABLE I
Combinators used in Cephalopode compiler and their reduction rules.

`I x`	→	`x`
`K x y`	→	`x`
`S f g x`	→	`(f x)(g x)`
`C f g x`	→	`(f x) g`
`B f g x`	→	`f (g x)`
`S′ f g h x`	→	`f (g x) (h x)`
`C′ f g x y`	→	`f (g y) x`
`B′ f x g y`	→	`(f x) (g y)`
`B⋆ f g h x`	→	`f (g (h x))`
`S′′ f g x y`	→	`(f x y) (g x y))`
`Ln e1 e2 ... en x`	→	`x e1 e2 ... en`
`Cn f e1 e2 ... en x`	→	`(f x) e1 e2 ... en`

Since Cephalopode is aimed to be an IoT processor, the code, i.e., the resulting combinator graph from the compiler, is finally written out as a ROM image. The translation from the graph to the ROM image is entirely straightforward.

TABLE II
Comparison fixed set vs. extended set of combinators.

Program	Nbr. combinators		Graph size		Reductions	
	Old	New	Old	New	Old	New
Factorial	13	5(3)	44	28	243	159
Dot product	34	13(7)	97	55	5928	3817
Matrix mult.	45	24(8)	134	88	7266	5174
Neural ntwk	92	53(16)	267	185	17276	12217
Min 3-D dist.	166	90(19)	405	240	4963	3639

VI. Evaluation methods

This section describes how the Cephalopode processor is evaluated in terms of speed, area, energy, and memory traffic. Since the processor is intended to be a low-energy IoT device we compare it to a RISC-V processor that one could imagine could fill that purpose [27] [28]. Preferably, since Cephalopode is a combinator graph reduction processor, we would like to also do combinator graph reduction on the reference machine. We use Augustsson's MicroHs [29] for this purpose. MicroHs is a simple Haskell compiler that compiles a Haskell program to a combinator graph, which can then be evaluated with the minimalistic MicroHs runtime system. We introduce MicroHs further in Section VI-A. In summary, we compare our processor that performs combinator graph reduction in hardware to a processor performing the same thing in software.

A. *MicroHs*

MicroHs is a simple Haskell compiler that compiles Haskell programs into combinator graphs, which can in turn be evaluated with the simple MicroHs runtime [29]. The language is an extended subset of Haskell-2010.

We are primarily interested in the MicroHs runtime system, which is written in C. Variables are handled internally with combinators, and primitive operations are built into the runtime system. One key thing that is not built into the runtime system is arbitrary precision arithmetic, which is implemented in

Haskell as a library rather than as runtime system primitives. As a result, the performance is significantly worse than a lower level implementation such as the one used in GHC. This will become apparent in subsequent sections.

The process of compiling MicroHs programs and executing them on the RISC-V machine is as follows. The MicroHs compiler *mhs* is first compiled normally for the host architecture (in our case x86-64) using GNU GCC. Next, we use this to compile our MicroHs program into a combinator graph file. This combinator graph file is bundled with the (slightly modified) MicroHs runtime system and compiled using the GNU GCC RISC-V tool chain. Next, we use the linker to produce ROM and RAM files that can be loaded into the instruction and data memories our simulated RISC-V machine.

GCC optimization level *-O1* is used to produce the RISC-V binary due to issues with stack overflow encountered when using higher optimization levels on MicroHs programs running on the RISC-V machine.

When performing measurements, we make sure to only measure during the execution time of *execio*. This avoids measuring initialization and tear down of the runtime system.

The corresponding process for Cephalopode was described in Section V. The program is implemented in fl, the meta-language of VossII and that is only syntactically different from the Haskell program used for the MicroHs version, and compiled into a ROM file as described in that section. That ROM file can then be loaded the simulated Cephalopode processor, which will then run the program to completion and terminate. As with the other example, only the graph reduction phase is measured.

Finally, to determine how much performance (energy and execution time) it costs to write our programs in Haskell, we also created handcrafted C code for all our benchmarks and compiled them and simulated the results on the same RISC-V processor.

B. Benchmark Programs

To create a representative benchmark suite for IoT applications, we selected one representative from each class in IoTBench [30], but restricted our choices to integer formats, since Cephalopode lacks a floating point unit at this stage. We then added a few more functional programming like programs to see if they behave substantially different. In all, we use 12 distinct programs. They are:

- *Factorial*: Computes $n!$ for $n = 10$.
- *Triple*: Computes 3^n for $n = 13$ using only additions.
- *MatrixAddConst*: Add a constant to all entries in a 10x10 matrix.
- *MatrixMulConst*: Multiply a constant to all entries in a 10x10 matrix.
- *Dot*: Compute the dot product of two 100 element lists.
- *Search*: Compute the sum of all elements in a list of 100 elements that satisfies an equality condition.
- *MinDistance*: Compute the square of the minimum distance in 3-D between a given location and 40 points.
- *Derivative*: Computes the difference between each two consecutive numbers in a list of 100 elements.
- *Conv*: Compute a 1-D convolution with a kernel of size 4, stride 1, and bias -10 over a vector with 50 elements.
- *NeuralNetwork*: Evaluate a neural network with (large) integer coefficients and with 3 hidden layers and ReLU activation functions.
- *Sort*: Sort a list of 40 integers.
- *MatrixMult*: Multiply two 10x10 matrices.

To illustrate the implementation of the programs in Haskell, see Listing 3 where we show the implementation of the fixed-size arithmetic *Factorial* program. All benchmarks use the same structure, where we compare the computation of our target expression to a precomputed answer. The reason for this comparison is to force a complete evaluation of the expression, and to aid in debugging.

```
module FixedFactorial(main) where
import Prelude

n :: Int
n = 10
answer :: Int
answer = 3628800

factorial :: Int -> Int
factorial n = if n <= 1 then 1 else n * factorial (n - 1)

main :: IO ()
main = (factorial n == answer) `seq` return ()
```

Listing 3. Implementation of the fixed-precision *Factorial* program.

Another example can be seen in Listing 4, where we show the arbitrary-precision version of the *MinDistance* program, that computes the square of the minimum distance in 3-D between a point and a collection of 40 obstacles.

```
module ArbMinDistance(main) where
import Prelude
import InputArbMinDistance

square :: Integer -> Integer
square x = x*x

sqdistance :: [Integer] -> [Integer] -> Integer
sqdistance (x1:y1:z1:r1) (x2:y2:z2:r2) =
    square (x1-x2) + square (y1-y2) + square (z1-z2)
sqdistance _ _ = 0

min_distance :: [[Integer]] -> [Integer] -> Integer
min_distance (x1:rem_locs) my_loc =
    let cur = sqdistance x1 my_loc in
    let rem = min_distance rem_locs my_loc in
    if( rem >= cur) then cur else rem
min_distance [] my_loc = 100

main :: IO ()
main = ((min_distance locs my_loc) == answer)
       `seq` return ()
```

Listing 4. Implementation of the arbitrary-precision *MinDistance* program. The input has been omitted for conciseness.

As mentioned before in Section III, Cephalopode integers are arbitrary-precision. Comparing only with fixed-size MicroHs would be therefore misleading in favor of MicroHs. On the other hand, only comparing with arbitrary-precision

MicroHs would also be misleading since its arbitrary precision code is quite poor in terms of performance[1]. Because of this, we evaluate each MicroHs program with both fixed and arbitrary precision.

Similarly, for the C-code versions of the benchmark programs, we evaluate it both using fixed integers and using an arbitrary precision arithmetic package from a production system [26].

C. Synthesis

Both processors are synthesized using the ASAP7 7-nm finFET predictive PDK and standard cell ASIC library [31] with Cadence Genus version 18.14. The library transistors are characterized at the TT corner at a 0.7-V supply voltage and a temperature of 25°C. With the processors intended as low-power IOT devices, we use an operating frequency of 100 MHz. We base our analysis on post-synthesis rather that post-place and route due to the complexity of the latter. For an apples-to-apples comparison, we consider post-synthesis to be sufficient.

Memories are modeled in behavioral Verilog and are not synthesized. They are modeled such that a read or write operation always take one phase. Since we also model their energy consumption as that of SRAM (described later in Section VI-D) the memories act essentially like L1-caches in a high-performance system.

It is possible to synthesize the RISC-V processor with and without features like super scalar mode, multiple issue, and with or without multiplication and division units. Since we prioritize low energy consumption, we turn off super scalar mode and use single issue. The multiplication and division units are however kept since Cephalopode has a multiplier, and we assume that the impact of the (inactive) division unit is negligible.

Once the post-synthesis netlists are generated, we execute the test programs described in Subsection VI-B in order to obtain cycle count, memory traffic and power consumption. We make sure to only measure for the combinator graph reduction (including allocation and garbage collection, should the latter run), taking no setup or tear down into account.

D. Memory energy model

As a final step, we need to take the power consumption of the memory into account. We use the following simple power model in order to approximate this. Previous work [32] found that a 8kB SRAM memory synthesized with the same standard cell library consumes roughly 0.25 pJ per bit per write operation and 0.23 pJ per bit per read operation. In our power model we scale these numbers with the width of each memory. Take for example the RISC-V RAM which is 32 bits wide. Therefore, we say that the energy consumption is $32 \cdot 0.25 \cdot 2 = 16$pJ for 2 write operations and $32 \cdot 0.23 \cdot 3 = 22.08$pJ for 3 read operations. Static power consumption for the memories is not considered since it is negligible for SRAM [33].

[1]Private communication, Augustsson, 2024.

VII. Evaluation

A. Results

Both processors are synthesized according to the methodology described in Section VI-C. The resulting netlists have gate counts of 48900 and 13500 for Cephalopode and the RISC-V machine respectively. Timing reports indicate that Cephalopode has a maximum clock frequency of about 420 MHz compared to about 300 MHz for the RISC-V machine. It is worth noting that no design effort has been put into limiting the size of Cephalopode at this time.

An important aspects of the Bifröst language is the ease in which protocols can be added between modules. We used this mechanism extensively to create separate (gated) clock domains. In practice, this meant we simply added `power = "clockgating"` to many of the protocol declarations used in Cephalopode. When we added clock gating originally, we saw a 40% decrease in energy consumption compared to before clock gating. Today, the current Cephalopode processor has 18 distinct clock domains and as a result, at run time most of the processor consists of "dark silicon" [34], drastically reducing the energy consumption.

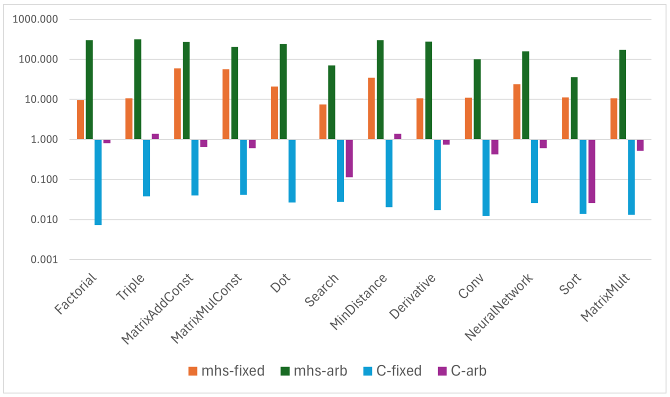

Fig. 3. Energy consumption relative to Cephalopode (lower is better).

In Fig. 3 we show the relative energy consumption compared with Cephalopode[2]. If we compute the geometric average we see that the C code using arbitrary precision arithmetic uses about half the energy of Cephalopode and the fixed integer C code uses about 48 times less energy. On the other hand, if we use MicroHs, we will use between 17 and 174 times more energy, depending on whether we use fixed integers or MicroHs's arbitrary precision package.

Although raw performance was not a primary objective in the design of Cephalopode, it is of course of interest. In Fig. 4 we show the total compute time for evaluating the benchmarks for our different configurations. Again, using the geometric average, we see that the C code using arbitrary precision arithmetic takes roughly the same amount of time as Cephalopode and the fixed integer C code runs about 22 times faster. On the other hand, if we use MicroHs, it will take

[2]Note that we use logarithmic plots. This makes data points within roughly 1% of Cephalopode indistinguishable.

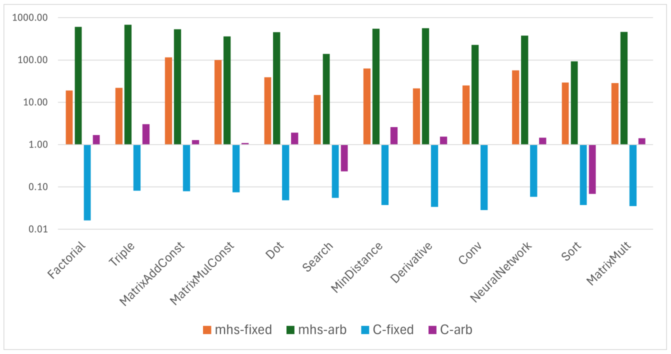

Fig. 4. Performance relative to Cephalopode (lower is better).

between 36 and 366 times longer, depending on whether we use fixed integers or MicroHs's arbitrary precision package.

In terms of power, the Cephalopode core is more efficient than the RISC-V machine, consuming 350 μW versus 450 μW on average. This is despite Cephalopode being 3.6 times larger in terms of cells.

B. Discussion

The results make it clear that Cephalopode is significantly outperforms MicroHs regardless of if arbitrary precision is used or not, and it performs about as well as the arbitrary precision C code. The fixed-precision C code is significantly better, which is hardly surprising.

Before drawing conclusions, we would like to raise a cautionary note. When comparing such different configurations and assumptions, it is very easy to make unfair comparisons. Looking at performance alone one could argue that fixed-precision C code is superior, but this misses out on the safety and productivity benefits of using a high-level functional language. As an analogy: a race car outperforms a normal car in terms of both speed and weight, but for numerous reasons is not an ideal choice for day-to-day use. One might also argue that MicroHs is a too primitive implementation to compare against, and that GHC would be the only good comparison. This is not yet possible, unfortunately, since GHC cannot emit RISC-V binaries. While GHC is indeed likely to do better on RISC-V than MicroHs—at least judging by their relative performance on desktop computers—the exact degree is uncertain, and comparison of Cephalopode to GHC on RISC-V is left until the latter becomes a reality.

Thus, the conclusions we draw from our benchmarking will need to be qualified rather carefully, and the associated assumptions stated explicitly.

With this proviso, we would like to draw the following conclusion: if the safety and convenience of using a high-level functional language is important, and we furthermore want to reduce, or even eliminate, most arithmetic overflow problems in our IoT developments, then something along the lines of Cephalopode appears quite attractive. It allows us to reach an energy consumption within a factor of two from a corresponding C program that also uses arbitrary precision arithmetic.

VIII. Future work

Avenues for further work fall into the categories of additional evaluation, enhancements to Cephalopode, and architectural exploration.

For evaluation, a more comprehensive suite of test programs would be ideal, particularly those that make extensive use of laziness and that are certain to trigger garbage collection in all architectures in question.

A direct comparison to a state-of-the-art functional language environment, e.g., one based on GHC, would be most illuminating, but must wait until such a system has been created whose runtime can fit on a tiny IoT processor.

Enhancements to Cephalopode include the addition of I/O and multitasking primitives, improvements to the compiler, the possibility for the multi-precision arithmetic unit to spill intermediate data into main memory in the case of very large integers, and an interface to allocate multiple nodes in a single clock cycle.

Today, the energy consumption of Cephalopode is dominated by the energy consumption of the memory (by a factor ranging from 5-10). Thus, it is clear that reducing the memory traffic in Cephalopode could have a dramatic impact on reducing its total energy consumption. Some of the design choices we made in Cephalopode, e.g., the use of back pointers rather than a stack or not using a simple 1-bit reference counting scheme to reduce the load on the garbage collector, should be revisited. It is our belief that there is significant reduction in energy for Cephalopode waiting to be implemented.

Some additional architectural possibilities to explore include the use of structured combinators [35], and support for unboxed integers.

IX. Conclusion

In this paper we have discussed the design, implementation, and evaluation of Cephalopode, a custom processor for combinator graph reduction for low-energy domains like IoT. We have provided a detailed description of the architecture and key design decisions, especially those relating to garbage collection, and a detailed description of the compiler. Cephalopode has been compared to a reference system with MicroHs running on a RISC-V core. We also compared the performance of C versions of the benchmarks running on the same RISC-V. With this in mind, we finally conclude that dedicated hardware for functional language execution in resource-constrained environments is both viable and efficient compared to imaginable alternatives.

References

[1] J. Leyden, "Samsung smart fridge leaves Gmail logins open to attack," Aug. 2015. [Online]. Available: https://www.theregister.com/2015/08/24/smart_fridge_security_fubar/

[2] A. Greenberg, "The Jeep Hackers Are Back to Prove Car Hacking Can Get Much Worse," *Wired*, Aug. 2016, section: tags. [Online]. Available: https://www.wired.com/2016/08/jeep-hackers-return-high-speed-steering-acceleration-hacks/

[3] Twitter, Email, and Facebook, "Hackers tapping home appliances to launch attacks," Oct. 2016, section: Science. [Online]. Available: https://www.sandiegouniontribune.com/news/science/sd-me-hackable-home-20161003-story.html

[4] S. Marlow *et al.*, "Haskell 2010 language report," *Available on: https://www. haskell. org/onlinereport/haskell2010*, 2010.

[5] P. Li and S. Zdancewic, "Encoding information flow in Haskell," in *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, Jul. 2006, pp. 12 pp.–16, iSSN: 2377-5459. [Online]. Available: https://ieeexplore.ieee.org/document/1648705

[6] S. L. Peyton Jones, *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc., 1987.

[7] L. Augustsson, "BWM," in *Functional Programming, Glasgow 1991*, ser. Workshops in Computing, R. Heldal, C. K. Holst, and P. Wadler, Eds. London: Springer, 1992, pp. 36–50.

[8] S. L. P. Jones, C. Clack, J. Salkild, and M. Hardie, "GRIP a high-performance architecture for parallel graph reduction," in *Functional Programming Languages and Computer Architecture*, G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmller, J. Stoer, N. Wirth, and G. Kahn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, vol. 274, pp. 98–112, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/3-540-18317-5_7

[9] M. Naylor and C. Runciman, "The reduceron reconfigured," in *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '10. New York, NY, USA: Association for Computing Machinery, Sep. 2010, pp. 75–86. [Online]. Available: https://dl.acm.org/doi/10.1145/1863543.1863556

[10] M. Scheevel, "NORMA: a graph reduction processor," in *Proceedings of the 1986 ACM conference on LISP and functional programming*, ser. LFP '86. New York, NY, USA: Association for Computing Machinery, Aug. 1986, pp. 212–219. [Online]. Available: https://dl.acm.org/doi/10.1145/319838.319864

[11] D. Weinreb and D. Moon, "The Lisp Machine manual," *ACM SIGART Bulletin*, no. 78, p. 10, 1981. [Online]. Available: https://dl.acm.org/doi/10.1145/1056737.1056738

[12] D. A. Turner, "A new implementation technique for applicative languages," *Software: Practice and Experience*, vol. 9, no. 1, pp. 31–49, 1979.

[13] W. R. Stoye, "The implementation of functional languages using custom hardware," University of Cambridge, Computer Laboratory, Tech. Rep., 1985.

[14] M. Scheevel, "NORMA: a graph reduction processor," in *Proceedings of the 1986 ACM conference on LISP and functional programming*, 1986, pp. 212–219.

[15] R. J. M. Hughes, "Super-combinators a new implementation method for applicative languages," in *Proceedings of the 1982 ACM symposium on LISP and functional programming*, 1982, pp. 1–10.

[16] J. Fairbairn and S. Wray, "Tim: A simple, lazy abstract machine to execute supercombinators," in *Conference on Functional Programming Languages and Computer Architecture*. Springer, 1987, pp. 34–45.

[17] L. Augustsson and T. Johnsson, "The chalmers lazy-ml compiler," *The computer journal*, vol. 32, no. 2, pp. 127–141, 1989.

[18] P. J. Koopman Jr and P. Lee, "A fresh look at combinator graph reduction," *ACM SIGPLAN Notices*, vol. 24, no. 7, pp. 110–119, 1989.

[19] S. L. P. Jones, "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of functional programming*, vol. 2, no. 2, pp. 127–202, 1992.

[20] Glasgow haskell compiler. [Online]. Available: https://www.haskell.org/ghc/

[21] M. Naylor and C. Runciman, "The Reduceron reconfigured," in *Proceedings of the 15th ACM SIGPLAN international conference on Functional Programming*, 2010, pp. 75–86.

[22] C. A. R. Melo, P. Liu, and R. Ying, "A platform for full-stack functional programming," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–5.

[23] T. Yuasa, "Real-time garbage collection on general-purpose machines," *Journal of Systems and Software*, vol. 11, no. 3, pp. 181–198, 1990.

[24] C. J. Cheney, "A nonrecursive list compacting algorithm," *Communications of the ACM*, vol. 13, no. 11, pp. 677–678, 1970.

[25] J. E. Hopcroft and J. D. Ullman, "Set merging algorithms," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 294–303, 1973.

[26] C.-J. Seger, "The VossII hardware verification suite," 2020. [Online]. Available: https://github.com/TeamVoss/VossII

[27] A. Waterman and K. Asanovic. (2017, May) The risc-v instruction set manual, volume i: User-level isa, document version 2.2.

[28] Risc-v. [Online]. Available: https://github.com/ultraembedded/riscv

[29] L. Augustsson, "Microhs," 2024. [Online]. Available: https://github.com/augustss/MicroHs/

[30] S. Chen, C. Luo, W. Gao, and L. Wang, "Iotbench: a data centrical and configurable iot benchmark suite," *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 2, no. 4, p. 100091, 2022.

[31] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm finFET predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, Jul. 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002626921630026X

[32] V. Vashishtha, M. Vangala, P. Sharma, and L. T. Clark, "Robust 7-nm SRAM design on a predictive PDK," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2017, pp. 1–4, iSSN: 2379-447X. [Online]. Available: https://ieeexplore.ieee.org/document/8050316

[33] P. Sandeep, P. A. Harsha Vardhini, and V. Prakasam, "SRAM Utilization and Power Consumption Analysis for Low Power Applications," in *2020 International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT)*, Nov. 2020, pp. 227–231. [Online]. Available: https://ieeexplore.ieee.org/document/9315558

[34] M. B. Taylor, "A landscape of the new dark silicon design regime," *IEEE Micro*, vol. 33, no. 5, pp. 8–19, 2013.

[35] C. Accetti, R. Ying, and P. Liu, "Structured combinators for efficient graph reduction," *IEEE Computer Architecture Letters*, vol. 21, no. 2, pp. 73–76, 2022.