

# The Reduceron: High Level Symbolic Computing on FPGA

## Part 2: Proposed Research

**Keywords:** functional programming; graph reduction; language-specific processor.

*Symbolic computing is a key technology with programs often written in high-level declarative languages. The Reduceron is an FPGA-based soft processor for executing such symbolic programs by graph reduction; the current prototype was designed in just 2–3 months as a case-study in the closing stages of Naylor’s PhD. Early results are sufficiently promising that we propose a 15-month feasibility study researching the potential of a special-purpose processor based on an advanced Reduceron. Results would be immediately applicable in FPGA-based systems and could inform the future design of a SPU (Symbolic Processing Unit) analogous to the current highly successful GPUs for graphics. Our proposal requires a comparatively modest investment of less than £100k.*

## 1 Background

**Symbolic computing** is a short-hand term for computing in which the structural combination of values is the dominant concern — rather than, for example, numeric calculation. It is a key enabling technology across a wide range of disciplines from linguistics to bioinformatics. Advances in computing technology itself rely on symbolic-computing tools to perform the most demanding tasks — eg. in circuit design, program analysis and optimisation, and automated verification.

It is possible to develop symbolic-computing applications in almost any general-purpose programming language. But some have much greater expressive power than others. In the higher-level declarative languages the effort and space needed to express solutions is a fraction of that needed in more conventional low-level languages. Programmers can “*think very big thoughts in one go*” [16]. Since the pioneering work on symbolic computing in LISP almost 50 years ago, there have been major design advances in higher-order, polymorphically-typed *functional languages* such as ML and Haskell (and the allied development of logic languages descended from Prolog). For really challenging symbolic applications, such languages are the only sensible choice.

**Current implementations** Compiling such high-level languages into efficient executable code is a huge challenge: they do not map easily onto hardware tuned for traditional low-level numeric benchmarks. Leading implementations such as the New Jersey ML compiler (SML-NJ) and the Glasgow Haskell Compiler (GHC) are large and highly sophisticated, the results of many years of team effort. These compilers are very impressive, but as they target conventional processors they have inherent limitations. They do not benefit as much as one might hope from increases in processor-speed because computation is typically *memory intensive*, governed by the rate at which data flows between processor and memory. Despite architectural advances, memory bandwidth is limited by essentially *serial* communication in *single units* — the *von Neumann bottleneck* [3]. Implementation techniques such as *graph reduction* require intensive construction of instances of function bodies in memory: building each instance requires serial execution of many instructions, not because of inherent data dependencies but because of the architectural constraints in conventional machines.

Functional languages also offer much scope for the parallel evaluation of expressions. However, on conventional architectures there is a high cost for operations such as locking and releasing expressions under evaluation. So the benefits of parallel evaluation are offset by significant communication overheads.

**Special-purpose machines** All this motivates the idea of computing machines specially designed to meet the needs of languages for high-level symbolic computation — much as GPUs are designed to meet needs in graphics. This is hardly a new idea. In the ’80s and ’90s there was a 15-year ACM conference series *Functional Programming Languages and Computer Architecture* (FPCA). In separate initiatives, there was an entire workshop concerned with graph-reduction machines alone [6], and a major computer manufacturer built a graph-reduction prototype [14]. But the process of constructing exotic new hardware was slow and uncertain, and there were some expensive failures. With major advances in compilation for ever bigger, faster and cheaper mass-market machines, the idea of specialised hardware design for declarative languages went out of fashion.

But now, *field-programmable gate arrays* (FPGAs) have greatly reduced the effort and expertise needed to build special-purpose hardware.

They are ideal for experiments with custom processing logic, providing thousands of small logic blocks that can be reconfigured at will using software tools. Similarly, memory architectures of all kinds can be constructed from large arrays of independent block RAMs. The *RAMP* project at Berkely [13] shows how much can be done using FPGAs for experimental *soft processors*. FPGAs may be seen as an advancing technology with continually improving performance, perhaps one day approaching the clock rates of modern PCs. Or, alternatively, as a tool for rapidly prototyping designs, with great accuracy, before they are manufactured as efficient, non-programmable ASICs. Both views motivate experiments in the design of special-purpose machines using FPGAs. Free of the constraints and intricacies of highly engineered conventional processors, the goal is to find fresh, minimalist solutions to architectural problems, exploiting opportunities for low-level parallelism. Special-purpose processors can be *not only faster but simpler* than conventional ones, with benefits such as fuller verification and lower energy consumption.

## Previous Work

**Reduction machines** There is a long history of work on computing hardware to provide improved support for high-level symbolic languages. Lisp Machines, for example, with hardware-support for dynamic memory management, emerged in the late '70s. More radical designs for reduction machines to evaluate functional programs appeared throughout the '80s and early '90s, and many were presented at conferences such as the FPCA series. Much, but not all, of the attention focused on possible implementations of *parallel graph-reduction*, helpfully surveyed by Clack [4]. Few special-purpose graph-reduction machines were actually built in hardware. Some, such as SKIM [15], were realised in microcode, but many were only ever evaluated by slow simulations in software.

The difficulty of outpacing advances in conventional processors was a serious drawback. One solution was to adopt commercial multi-processors as targets, building support for parallel graph-reduction into clever run-time systems: Buckwheat [7] and the  $\langle \nu, G \rangle$ -machine [2] are good examples, both with shared-memory hosts. Another solution was to use stock processors as components in a custom machine: GRIP [12] was the most sophisticated example, a *virtual shared memory* multi-processor for parallel graph reduc-

tion. Some very promising results were achieved, but even these implementations were overtaken by the advancing technology of uniprocessors.

Augustsson's design for the Big Word Machine (BWM) [1] used wide, parallel memories to increase performance. The BWM is a serial graph-reduction machine but exploits *low-level parallelism* to make each reduction fast. It is minimalist in approach. A fast switch allows complex stack rearrangements to be done by a single instruction. Words wide enough for four pointers make it quick to build and retrieve expressions in heap memory. The BWM does not uniformly update the heap with results of reduction: if a computation is known not to be shared, unnecessary heap accesses can be avoided. The BWM was never actually built: some software simulations were done, but not enough to determine what the absolute performance would be.

**Reduceron prototype** In the closing stages of Naylor's PhD work on the application of functional languages to hardware design, he needed a case study and chose a minimalist implementation of graph reduction on an FPGA. In 2-3 months a prototype design was completed, and dubbed *The Reduceron*. It is based on a very simple abstract machine with just five instructions. It has several features in common with the BWM, though Naylor was at first unaware of Augustsson's work. The FPGA implementation exploits parallel memory access, separating not only stack-space but also heap-space from code memory. It has a fast switch for stack manipulation. It uses a mechanism for wide (dual-port quad-word) operations to speed the fetching of applications, access to the stack and the instantiation of function bodies. Less memory is wasted than in the BWM as data need not be aligned on four-pointer boundaries. Although there is no support for update avoidance, application spines can be instantiated on the heap and stack simultaneously, in a single clock cycle. The prototype compiler to Reduceron code performs no optimisation, only the essential transformations to make code generation possible. Even so, the Reduceron on a slow FPGA (clock-speed 91.5MHz) consistently outperforms the fastest byte-code implementations on a PC (clock-speed 2.8GHz, or 30 times that of the FPGA). A fuller account of the Reduceron prototype has been accepted for publication [11].

## 2 Programme & Methodology

### Aims and Objectives

*We propose to explore the further potential of a machine like the Reduceron. Our broad aim is to advance understanding of how FPGA technology can most effectively be used as hardware infrastructure in a graph-reduction implementation of a functional language. Our overall objective is to demonstrate advances in the design and implementation of the Reduceron, at several different levels of computational abstraction, with the combined effect of a leap forward in performance.*

Specifically, our objectives include advances at the following four levels, from highest to lowest:

1. *compiler optimisation* when translating a core functional language to Reduceron code;
2. *high-level parallelism*, reducing more than one function application simultaneously;
3. *low-level parallelism*, increasing memory utilisation and the amount of work performed in each clock cycle;
4. *critical-path optimisation*, restructuring the Reduceron to shrink the critical path and increase clock-speed.

In comparison with the prototype, we aim to double performance by compiler optimisations alone, to double it again by a combination of high-level and low-level parallelism, and to gain a further 50% speed-up by critical-path optimisation. If we succeed in these aims, then for symbolic applications our FPGA-Reduceron will outperform even the most advanced optimising compilers on modern PCs. Potentially, transferring the design to an ASIC-Reduceron would give an order of magnitude speed-up. In addition to such performance gains there are also the eventual benefits of architectural simplicity for verification, predictability and lower energy consumption.

### Methodology

Our implementation methods will continue to exploit functional-language technology. The VHDL net-list description of the Reduceron is generated from a functional program using a variant of the Lava combinator library [5] for hardware-description in Haskell. Among the advantages of using such a high-level description, we can perform automated property-based testing using established tools such as QuickCheck (again see [5])

and our own more recently developed tools such as Reach [10]. The mapping of components to specific FPGA resources is handled by standard Xilinx tools.

The compiler for Reduceron uses the core-language and associated tools from the *Yhc* byte-code compiler. This avoids needless duplication of effort at the front-end and gives the possibility of adapting recent advances in *Yhc* for use with Reduceron – see *Compiler Optimisation*.

The prototype Reduceron is implemented on a Xilinx Virtex-2 FPGA. It uses under half of the 10k logic slices available, so there is plenty of room for additional circuitry — but only so long as there is a single Reduceron on the chip. Memory capacity is already stretched to the limit. Cascaded block RAMs provide heap memory of just 32k 18-bit words and an auxiliary space of 12k words for use during stop-and-copy garbage collection. Further parallel memory structures provide 4k words each for a Reduceron-coded program and two stacks for arguments and update pointers during reduction. Together these structures fully occupy all 56 1k-word block RAMs available.

Advances such as parallel reduction, and the need to accommodate larger programs, require a bigger machine. So we intend to begin by adapting the prototype Reduceron design for the similar Virtex-5 FPGA. It is almost twice as fast — our prototype clocked at 91.5MHz on the Virtex-2 would run at around 160MHz on the Virtex-5 — and offers greater capacity in both logic and memory. The exact model will be determined by what we can obtain in the UK at the time but, for example, the SX95T would give 50% more logic and ten times the memory.

### Timeliness and Novelty

**The need for symbolic computing** is increasing with the emergence of fields such as bioinformatics and rising ambitions for exact modelling and verification of software. High-level declarative languages are essential tools for successful expression of advanced programs in these areas. Efficient implementation of such languages on conventional machines usually depends on complex compilers and run-time systems. We propose to research the feasibility of using a different kind of machine to obtain higher performance despite simpler processes both at compile-time and at run-time.

**The advent of multi-core processors** has re-awakened widespread interest in how language implementations can exploit shared-memory parallel computation. Work on parallel graph reduction has recently been revived in this context [8]. But conventional processors reflect an engineering commitment to highly-tuned forms of low-level parallelism that reward localised sequential computing, so the new high-level parallelism and the established low-level parallelism are not that easily combined. For example, the high cost of instructions for locking and releasing graphs under evaluation has prompted complex schemes for lock-free parallel reduction with techniques to bound the amount of duplicated work. The Reduceron is a comparatively simple machine giving us an opportunity to explore the design space for effectively combined high-level and low-level parallelism.

**The development of FPGAs** and of tools for the high-level compositional expression of low-level hardware configurations, makes feasible the rapid exploration of design alternatives for radically different machines. Although various designs for graph-reduction machines have been around for years, most of them are untried in practice because of the time and effort needed for hardware-based experiments. We are not aware of any other researchers who have built a graph-reduction processor on FPGA capable of running programs compiled from a standard functional language.

**The results from the prototype** are sufficiently promising, given its very short development time, to warrant further investigation. Naylor is the obvious person to pursue the work. He is motivated and he is available.

## Programme of Work

We outline some of the main lines of work we intend to pursue at each of the four levels. Some dependencies between levels are noted.

**Compiler Optimisation** The prototype compiler from Yhc-core programs to Reduceron code uses a simple and uniform translation scheme without any attempt at optimisation. We propose to investigate the combined use of *supercompilation* and *wide body fitting* as optimisation techniques. We also plan to add specialised Reduceron support for building and examining data structures.

The idea of any compiler optimisation is to do more at compile-time in order to do less at run-

time. A *supercompiler* takes this idea to a natural conclusion: it tries to evaluate the program at compile-time, expressing the remaining work as a *residual program*. The ideas are similar to those in *partial evaluation* and *program specialisation*, but no input data is required, only a program. Few supercompilers have been built, but an experimental supercompiler for Yhc-core programs has recently been developed in our group [9]. It is effective as an “accelerator” used in conjunction with GHC (already a highly optimising compiler). Although Reduceron is a very different target, we expect to obtain significant gains from an appropriately adapted version of the supercompiler.

The current supercompiler leaves inlining in the residual program to GHC which uses various heuristics. For the Reduceron the optimal inlining decisions can be informed by exact simple formulae: *eg.* the prototype takes  $3 + \lfloor n \div 8 \rfloor$  clock cycles to reduce an application of a function with an  $n$ -word body, the divisor 8 representing dual-port quad-width instantiation steps. We also have an exact bound on program-code memory, and we might as well fill it. Lambda lifting after encoding data types as functions often introduces a new function definition for each case alternative, breaking function bodies into smaller pieces. Examining the dynamic distribution of body sizes for a sample of programs running on the prototype, from 35%–50% are less than 4 words. So width-tuned inlining could be very effective. The cost model might be further refined by *strictness* information, or profiles of sample runs.

**High-level Parallelism** Our main goal under this heading is to achieve *parallel graph reduction*.

We propose to implement a *virtual shared-memory* graph reducer by combining two or more evaluators on a single FPGA — a *multicore Reduceron*. Schemes based on shared heap memory are simpler and have in the past tended to give better results, but considerations of low-level parallelism and memory resources demand some partitioning of memory to avoid contention between evaluators. Stack memories should be distinct, not segmented in the heap. Code memory should also be partitioned to avoid contention, perhaps informed by a simple static analysis or by profiling information. Another use of spare memory capacity would be code replication.

On the Reduceron, our architectural constraints are comparatively few and simple, so we hope for fast machinery to implement locks and blocks on heap expressions under evaluation. We shall per-

form only strictly needed computations in parallel, avoiding the complexities of speculative evaluation — though in future work, speculation could be another fruitful source of parallelism.

On-the-fly garbage collection is another significant challenge for high-level parallelism. We have experience developing on-the-fly collectors, and do not rule out some effort in this direction, but it will probably be beyond the scope of this short project.

**Low-level Parallelism** The overall objective under this heading is to increase the degree of parallelism by *improved memory utilisation*.

Part of the work here is just the machine-level counterpart of our plans to improve compilation: the high-level motivation for in-lining is to decrease the number of reductions; a lower-level perspective is that larger function bodies make better use of available bandwidth. We propose to extend the Reduceron instruction set so that fewer function bodies need to be split, and constructors simply rearrange the stack, with no need to instantiate them in the heap.

Further observations suggest other ways to increase utilisation of memories. Primitive evaluation involves neither heap memory nor code memory. Memory is buffered to avoid excessive cycle time, but in consequence for much of the time memory is only accessed every other cycle — there is pipelining during the instantiation of bodies, but plenty of scope for pipelining elsewhere. Another limiting factor in the prototype is that most applications are only one to five words in size, and indirections used to achieve sharing are only one word wide. Possible solutions include building combinator spines directly on top of redex roots, and the use of shallow trees instead of flat sequences for representing function applications.

**Critical-path Optimisation** The objective at the lowest level is to minimise the critical path so that *higher clock-rates* can be achieved. We propose to extend synthesis tools to give improved resource feedback, and use this information to refine the high-level description of the Reduceron.

The clock-rate for the prototype Reduceron is 91.5MHz on the Xilinx Virtex-2 FPGA. In comparison, a carefully optimised design for a small 8-bit processor, the Xilinx PicoBlaze, can be clocked at 173.5MHz on the same device. A factor of two between the clock-speeds of a prototype and of a refined product may not be too surprising. However, for all the differences between the

two architectures, here is a strong indication that a much higher clock-speed may be attainable for the Reduceron.

Little attempt has been made to ensure that the prototype design is realised by an FPGA implementation with a high clock rate. The functional-language description in Haskell with Lava computes only a net-list in VHDL, and resource allocation is devolved to a generic Xilinx tool. The tool does give information about the critical path, but only at net-list level; there is currently no facility to trace the critical components in the high-level description. We propose to extend the definitions of Lava operators to allow descriptive labels to be attached to wires. In this way wires appearing in the final netlist will contain information about their origin or purpose, giving meaning to the critical path reported by the synthesis tool.

Shortening the critical path is typically achieved by removing logic or inserting registers as appropriate. For example, the combinatorial path for the logic implementing memory writes is currently rather long. It includes rotation logic to implement quad-word memories and multiplexors to combine the write requests of each Reduceron instruction. It could be shortened by inserting a register between the instruction logic and the quad-word logic. Alternatively, instead of using quad-word memories, a larger word-size could be used — despite the disadvantage that space may be wasted due to alignment restrictions.

## Risk and Project Management

Our multi-level research programme might seem rather ambitious for a short project. As we cannot be sure in advance of the relative importance of the different levels, and the interactions between them, all must be included. Given our experience of working on the prototype, we think we can achieve worthwhile results even in a short time. At the very least, we are confident that significant performance gains can be made, and that we can discover in detail the strengths and weaknesses of the Reduceron approach at each level. As an additional safeguard, the planned programme will yield first a refined sequential reducer exploiting low-level parallelism, only then proceeding to work on a parallel reducer.

As to management, this is a small project. We have already established a very fruitful working relationship during Naylor's PhD. The specific mechanisms for managing the project will be weekly discussions and quarterly reviews.

### 3 Relevance to Beneficiaries

The immediate beneficiaries of our work will be researchers and developers working to match hardware architectures with the needs of modern programming languages. It will be of special relevance to those working with functional languages, and the problems of effectively combining low-level and high-level parallelism.

Our work will also be of interest to the verification community. The relative simplicity of the Reduceron compiler and run-time structures make them more amenable to proof than complex compilers, run-time systems and processors now in use.

The intended long-term benefits are improved levels of performance, verification and energy efficiency for a wide range of symbolic computing applications.

### 4 Dissemination and Exploitation

In a 15-month feasibility study. We do not expect to achieve results ready for immediate commercial or industrial exploitation.

**Web Site** A project web site will provide a continually updated overview of our work and a repository of information relating to the project.

**Published Papers and Talks** We intend to publish at a mixture of international conferences and workshops including CC, HFL, ICFP and IFL. (Conveniently, CC and HFL are both due to be held in York in Spring 2009 as part of the joint ETAPS conferences.) We shall also seek opportunities to present our work in talks at less formal meetings such as FitA in the UK. Towards the end of the project we aim to submit a more definitive account of the work to a journal.

**Open Sources** We shall continue to make sources for the Reduceron, and its associated compiler and interpreter, freely available for downloading.

### References

- [1] L. Augustsson. BWM: a concrete machine for graph reduction. In *Functional Programming, Glasgow 1991*, pages 36–50. Springer-Verlag, 1992.
- [2] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the  $\langle \nu, G \rangle$ -machine. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 202–213. ACM Press, 1989.
- [3] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *CACM*, 21(8):613–641, 1978.
- [4] Chris Clack. Realisations for non-strict languages. In *Research Directions in Parallel Functional Programming*, pages 149–187. Springer, 1999.
- [5] Koen Claessen. Embedded Languages for Describing and Verifying Hardware. PhD Thesis, Chalmers University of Technology, 2001.
- [6] Joseph H. Fasel and Robert M. Keller, editors. *Graph Reduction, Proceedings of a Workshop*. Springer LNCS 279, 1987.
- [7] Benjamin Goldberg. Buckwheat: Graph reduction on a shared memory multiprocessor. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 40–51. ACM Press, 1988.
- [8] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Proc. ACM SIGPLAN workshop on Haskell*, pages 49–61. ACM, 2005.
- [9] Neil Mitchell and Colin Runciman. A supercompiler for core Haskell. In *Implementation and Application of Functional Languages (IFL 2007, Revised Selected Papers)*. Springer LNCS (to appear), 2008.
- [10] Matthew Naylor and Colin Runciman. Finding inputs that reach a target expression. In *SCAM '07: Proc. 7th IEEE Intl. Conf. on Source Code Analysis and Manipulation*, pages 133–142. IEEE Computer Society, 2007.
- [11] Matthew Naylor and Colin Runciman. The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. <http://www.cs.york.ac.uk/~mfn/reduceron.pdf>, to appear in *Implementation and Application of Functional Languages (IFL 2007, Revised Selected Papers)*, Springer LNCS, 2008.
- [12] S. L. Peyton Jones, C. Clack, and J. Salkid. High-performance parallel graph reduction. In *Proc. Parallel Languages and Architectures Europe (PARLE'89)*, pages 193–206. Springer LNCS 365, 1989.
- [13] RAMP: Research Accelerator for Multiple Processors. <http://ramp.eecs.berkeley.edu>, 2008.
- [14] Mark Scheevel. NORMA: a graph reduction processor. In *Proc. ACM Conf. on LISP and Functional Programming*, pages 212–219. ACM, 1986.
- [15] William Stoye. The Implementation of Functional Languages using Custom Hardware. PhD Thesis, University of Cambridge, 1985.
- [16] David A. Turner. The semantic elegance of applicative languages. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA'01)*, pages 85–92. ACM Press, 1981.