Development of a customized processor architecture for accelerating genetic algorithms ⋆

Nikolaos Kavvadias *, Vasiliki Giannakopoulou, Spiridon Nikolaidis

Section of Electronics and Computers, Department of Physics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece

Abstract

In this paper, a new programmable RISC processor architecture named VGP-I is proposed, aiming to the acceleration of genetic algorithms in embedded systems. Compared to other GA engines, the VGP-I specification defines a compact instruction set supporting multiple operator types, with scalable instruction encodings, programmer-visible and auxiliary registers and optional extensions. Apart from the programmable accelerator approach, VGP-I instructions have been tightly integrated to the Nios II soft-core processor as well. For performance assessment, a cycle-accurate reference VGP-I model has been developed while VGP-I subsets have been realized on a prototype microarchitecture and as Nios II custom instructions, both verified on programmable logic. Performance improvements on the execution of genetic operators are typically at the level of two orders of magnitude with application kernels written in ANSI C being accelerated by about 20× due to the usage of GA instruction set extensions.

Key words: Embedded systems, Field-programmable gate arrays, Genetic algorithms, Application-specific processors, Hardware description languages
PACS: 89.20.Ff

⋆ This work was supported by the General Secretariat of Research and Technology of Greece and the European Union.

* Corresponding author.
Email addresses: nkavv@physics.auth.gr (Nikolaos Kavvadias), vgian@physics.auth.gr (Vasiliki Giannakopoulou), snikolaid@physics.auth.gr (Spiridon Nikolaidis).

1 Introduction

'Genetic algorithms' (GAs) are used as solution space search algorithms based on the mechanisms of natural selection [1]. Their strength is that detailed knowledge for solving the confronted problem is not necessary; in most cases the larger part of the GA can be kept application-agnostic with only the process of evaluating candidates from a solution pool ("population") needing to be aware of solution quality metrics. A subset of the population is extracted by means of a selection procedure. A reproduction operator (as crossover and mutation [2]) is applied to the individuals to produce offspring whose performance is evaluated by a fitness function and a survival mechanism may be used for advancing the population to the next state. Finally, the fitness values are also used to determine whether the search process has finished, i.e. an acceptable solution has been found.

The major strength of GAs is that for most applications, components with the exception of the objective function can be reused across different optimization problems. For this reason, each GA should be carefully profiled in order to determine the computational bottlenecks of the entire algorithm. It is frequently stated that for most problems, fitness evaluation dominates over the other contributors. This may be true for certain applications, i.e. DNA code word library synthesis for large sequences [3], however the proportion of each contribution to the overall evaluation time is affected in many cases by factors stemming out of the implementation approach (hardware or software) and its complexity. For example, if a GA is implemented in software on a typical embedded processor (e.g. DLX [4]), many useful features for GA computation are missing: bit-oriented instructions for manipulating either bitstrings or arbitrary-sized integers for crossover and mutation operations, fast pseudo-random number generation [5, 6] and useful forms of data parallelism as SIMD [7] or systolic arrays [8]. The missing features impact the execution time of the different GA components, while even if the fitness function is heavily accelerated, Amdahl's law mandates that the remaining GA constituents will be the performance bottlenecks for the overall procedure.

In this paper, a genetic algorithm processor architecture is introduced that can be used in the development of embeddable programmable GA cores. The GA processor architecture, named VGP-I (first generation of a Very applicable Genetic Processor), features scalable encodings and considers configurable implementations of genetic operators as specialized functional units (SFUs) supporting a wide range of crossover, mutation and selection types. These operators provide fundamental functionalities shared across different GA schemes and can be reused for two basic representations of genotype data: bitstrings and integers. The most significant features of VGP-I are the introduction of a custom instruction set and RTL semantics for efficient GA computa-

tion, on-the-fly adaptation of the type of any genetic operator with zero-cycle latency and optimal utilization of embedded memory blocks (block RAMs) found in current Field-Programmable Gate Array (FPGA) implementation devices. Further, to examine an instruction extension approach, a VGP-I subset has been implemented as custom instructions for the Altera Nios II soft-core processor [9].

The main contribution of this paper is the development of an ASIP (Application-Specific Instruction set Processor) targeting the application-agnostic content of different GAs. The focus of this work is two-fold:

- defining the programmers' and architects' view of a GA processor inheriting advantages of the ASIP paradigm: ability to target different applications, flexibility to cope with evolving requirements, easiness of maintenance without sacrificing too much performance against a hardwired ASIC solution.
- definition of a GA engine architecture significantly different from other approaches [6, 10]:
  - function-level instructions as key to achieve significant performance improvements.
  - Hardware reuse enabled by feature sharing across operators.

The remainder of this paper is organized as follows. Section 2 overviews previous research regarding the design and applicability of other hardware GAs. In Section 3, the VGP-I architecture is described in detail. Details on VGP-I implementations as an ASIP prototype and as instruction set extensions to a soft-core processor are discussed in Section 4. Performance comparisons on the acceleration of GAs running on VGP-I architectures against optimized software implementations running on general-purpose embedded RISC processors are given in Section 5. Finally, Section 6 summarizes the paper.

2 Related work

In recent work, a number of hardware implementations of genetic processing engines has been proposed which can be classified into two main categories: dedicated hardware GAs [11–17] and programmable processor GAs [6, 10].

The early implementations of hardware GAs employed FSM-based control, a single type for each genetic operator and fitness evaluation units for specific problems. SPGA [12] and HGA [11] are two of the most popular GA engines conforming to these characteristics. The SPGA realizes a hardware implementation of the simple genetic algorithm (SGA) [1] with fixed operator types (single-point crossover, single-bit mutation and routine-wheel selection) for the traveling salesman problem (TSP). In a similar sense, the HGA [11]

is a dedicated accelerator that can be used as a loosely-coupled coprocessor in an embedded system. In the H^3 GA machine [13], a crossover and a mutation unit have been used, which can perform uniform crossover and one type of mutation, respectively, with predefined probability. Instead of SGA, the non-generational steady-state genetic algorithm (SSGA) was used to eliminate pipeline stalls induced by the nature of SGA. More recently, a GA machine has been presented in [17] organized as a 6-stage pipeline targeting a 36-residue protein folding and a set-covering problem modeled via a survival-driven SSGA. Only random selection can be executed, however the built-in survival mechanisms compensate for the inefficiency of random selection. The main drawback of this approach is the lack of programmability which infers reduced flexibility and reuse capability in order to be used for varying environments that require reconsideration of operator probabilities, the specific operator types or the overall GA policy (choose a different algorithm).

Some works have proposed less rigid forms of hardware GAs implementing alternative genetic operator types, since the choice of the specific selection, reproduction, survival and termination criterion mechanisms indirectly affects the execution time of a GA. For example, if an inappropriate crossover technique is used (e.g. that induces 'crowding') the entire GA might get stuck at suboptimal solutions, hindering convergence and increasing execution time. This can be overcome by online (on-the-fly) adaptation of the operator types [18,19]. Indeed, a flexible crossover module can be found in [18], still only supporting 2-point [20] and uniform [21] crossover. The combination of reproduction operators with desctructive (uniform crossover) and non-destructive (2-point crossover) nature of potentially favourable schemata can assist in maintaining good solutions and effective sampling of the solution space.

A few recent works have identified the need for parameterized or programmable genetic algorithm processors [6,10] and explicitly stated the unavailability of turn-key GA processor cores [3]. A truly programmable GA processor can be found in [10] where an 8-bit stack-based microcoded design has native support for operation-level primitives such as extracting/setting bit sets. Selection and fitness evaluation are not taken into account in their design, while there is complete absence of any performance evaluations regarding the processor's efficiency. It seems that this processor is applicable in a context with low-performance requirements where other issues such as chip area are of primary importance.

Instruction-set extensions for accelerating GA execution have also been added to the DLX RISC processor [4] in [6]. The primary aim in [6] is the integration of operation-level extensions to the programming model of an existing processor. The added instructions provide support for simple bit-oriented operations, pseudo-random number generation, and SIMD processing for the GA-related operations. Up to 90% performance speedup is observed for the implementa-

tion of genetic operators as software on DLX-GA against the unaugmented DLX. For example, 2-point crossover requires 11 instructions on DLX-GA compared to 138 on DLX. A 3× application-level speedup is reported for a simple GA problem when using the instruction set extensions. While the capability of executing a wide range of GAs is preserved, the maximum possible performance gains have not been obtained.

The GA processor architecture in our work overcomes many of the aforementioned limitations since we follow an ASIP-based approach capable of delivering better performance. Further, although most of the previous GA engines have been implemented on FPGAs, it is not clear that they were particularly optimized for these devices. Also, the majority of these processors have been designed for solving a specific problem and the applicability of an instruction set at least for the reusable part of the GAs has not been examined with a few exceptions [6, 10]. The potential of treating GA operators as complex custom instructions and mapping them on the SFUs of a programmable processor has not been examined elsewhere.

3 The VGP-I architecture

3.1 Overview

A GA processor supporting an appropriate programmer's model should be able to execute whole self-contained GA kernels, and be usable as a coprocessing element in context of an embedded system for servicing larger applications. The VGP-I specification has been designed to fulfill such requirements. Currently, VGP-I supports 6 different crossover, 4 mutation and 4 selection operators. In this first incarnation, there are no conditional control flow operations in VGP-I; these are assumed to be executed by the control-plane processor in the system, which is a valid assumption for tightly-coupled application accelerators to contemporary processor cores as ARM [22], MIPS [23], and LEON [24].

The VGP-I enables configurability of operator implementations and runtime selection of different allele lengths within a lean ISA. In the following subsections, the VGP-I ISA is described in detail.

3.2 Instruction formats

The VGP-I uses a compact GA-oriented instruction set with scalable RISC encodings while the genetic operators are supposed to be organized in SFUs.

Table 1
Constraining parameters of the VGP-I architecture.

Parameter	Symbol	Useful range
Instruction word	I_w	16-32
Population size	p_s	Power-of-2 in {2,1024}
Num. chromosomes	n_c	1-16
Num. alleles/chromosome	n_a	Power-of-2 in {1,64}
Allele size	al_s	Power-of-2 in {1,64}
Probability resolution	p_r	$1/2^n$, n=0-15

For specifying the constraint parameters of the VGP-I architecture, we have taken account limitations due to the characteristics of reconfigurable implementation devices such as FPGAs. Having in mind block RAM configuration modes, the range of these parameters has been extracted: 16-bit encodings are minimal, while there is provision for population sizes up to 1024 (single-chromosome) individuals. The configuration parameter set is summarized in Table 1.

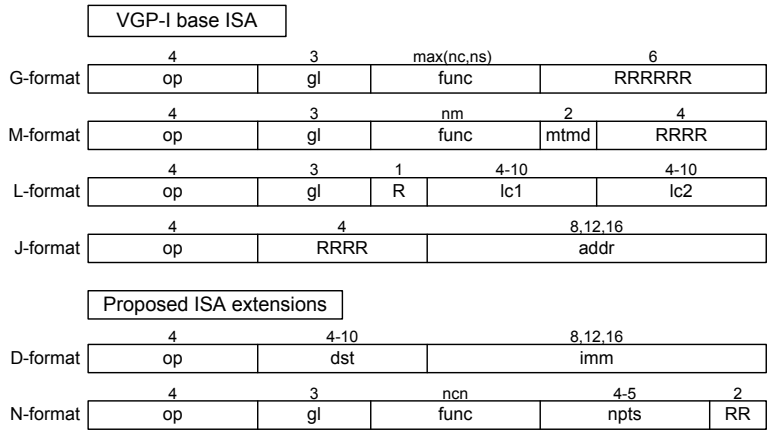
The fixed portion of the VGP-I ISA utilizes four different formats as depicted in Fig. 1. The $lc1$ and $lc2$ fields of the L-format are interpreted as chromosome or fitness register file indexes with the exception of fitness evaluation instructions. In this case, $lc1$ determines the actual fitness function and $lc2$ the appropriate fitness mode (evaluation of parent 1, 2 or both). Also, instructions adopting the G-format can use up to two register operands with implicit addressing. For example, crossover operators have two source and two destination operands with the same addresses.

3.3 The VGP-I instruction set

The fixed portion of the VGP-I ISA consists of 24 instructions, which are described alongside their interpretations and instruction format classifications in Table 2. The semantics of the instructions realizing genetic operators have been inspired by corresponding ANSI C descriptions found in an open-source GA library [25].

3.3.1 Crossover instructions

Six different types of crossover operations have been specified in VGP-I. Typical crossover operators accept two parent chromosomes and exchange their genes (comprised of a single allele) in a specified manner; two offspring chromosomes are produced as the result.



Name	Explanations of instruction fields
op	Primary opcode
gl	Gene (allele) length
func	Secondary opcode
mtmode	Mutation mode (father, mother or both)
lc1 | lc2	Chromosome/fitness register address
dst	Any register address (special registers are treated as mapped on a virtual register file named SRB
addr	PC-relative/absolute program address
imm	{8,12,16}-bit immediate
npts	{4,5}-bit immediate (currently only for n-point crossover)

Name	Instruction encoding parameters
nc	Max. number of crossover types
nm	Max. number of mutation types
ns	Max. number of selection types
ncn	Max. auxiliary crossover types

Fig. 1. VGP-I instruction formats.

Single/double/n-point crossover (CRC[S|D|N]P) **[20]:** Alleles of the parent 1 chromosome up to an odd-ordered crossover point are copied to the offspring 2 chromosome and correspondingly, alleles from the parent 2 chromosome are copied to the offspring 1 chromosome. For the remaining alleles situated from this point to the subsequent even-ordered crossover point (or to the chromosome end for CRCSP), those originating from parent 1 are copied to offspring 1 chromosome and parent 2 alleles are copied to offspring 2. The crossover points are randomly selected in the range $\{0, n_a\}$. n-point crossover requires a number of iterations to complete equal to n.

Mixing crossover (CRMX): CRMX mates two genotypes by mixing parents chromosomes. The decision to which offspring the parent chromosome will be copied, is taken by a random bit. The same type could be used both as a form of swap mutation [26] (EXNG instruction) or as an active NOP (COPY) if the decision bit is preset in the instruction encoding.

Allele mixing crossover (CRAMX) **[21]:** Parent alleles are randomly exchanged to form the two offspring chromosomes. If it is permitted to select the same chromosome as both parents, swap mutation is obtained as a degenerate version of CRAMX as well.

Table 2
The VGP-I instruction set.

Mnemonic	Format	Explanation
		Crossover instructions
CRC[S|D|N]P	G	Single/double/n-point on chromosomes
CRMX	G	Mixing
CRAMX	G	Allele mixing (uniform)
CRMN	G	Mean
		Mutation instructions
MTSP[D|R]	M	Single-point drift/randomize
MTMP	M	Multi-point
MTAP	M	All-point
		Selection instructions
SLRN[O|T]	G	Selection of one/two random address(es)
SLB2[O|T]	G	Binary tournament for one/two address(es)
		Other mandatory instructions
FIT	L	Fitness calculation for specific problems
TCN	L	Termination criterion
SURV	G	Apply a replacement strategy for offspring
LDRNG	L	Initialize one/two individuals with random values
LDEXT	L	Load data for an individual with I/O instructions
EVALP	G	Evaluate probability for crossover/mutation
SETRNGR	J	Set probability resolution to PRNGR
JMP	J	Jump unconditional
HALT	L	Force the processor to halt mode
NOP	L	No-operation
		Optional VGP-I instructions
LB[C|F]RB	D	Load byte (high/low) to CRB/FRB register
LBS	D	Load byte (high/low) to special register

Mean crossover (CRMN) **[27]:** This type mates the genotypes by averaging respective alleles of two parents. If the corresponding alleles' sum is positive, $sum/2$ is assigned to the offspring 1 allele and $(sum + 1)/2$ to the offspring 2 allele. On the contrary, if the parent alleles sum is negative, the offspring 1 allele obtains $(sum - 1)/2$, while the offspring 2 allele $sum/2$. This procedure is executed for all chromosomes of an individual.

3.3.2 Mutation instructions

Mutation instructions provide the means for asexual production of new offspring [2]. Four different types are currently implemented, all of which accept one parent chromosome and modify a randomly selected allele subset to generate a single offspring chromosome.

Singlepoint drift mutation (MTSP[D|R]): A random allele is chosen from a random chromosome, and is either incremented or decremented by one (MTSPD) or assigned to a random value (MTSPR), produced by a random number generator (RNG) e.g. an LFSR-based one [28]. Following MTSPD, overflow checks assure that the mutated allele is assigned the minimum value (modulo arithmetic).

Multipoint (MTMP): Executes the MTSPD type, for all alleles of an individual's chromosomes.

Allpoint (MTAP): Mutates each allele, according to the value of a three-state random variable. The allele will either increment its value by one, decrement by one, or remain unmodified.

3.3.3 Selection instructions

Regarding selection, two different types are supported with two alternative modes generating one or two individual addresses. The parent individual addresses are stored in dedicated registers (FCRA and MCRA in Table 3).

Select one/two random (SLRNO/SLRNT): Select-one-random loads FCRA with a randomly generated value, while for select-two-random both the FCRA and MCRA registers are updated by random addresses. The select-one-random selection type is appropriate to the case of mutation-only genetic algorithms.

Select one/two best-of-two (SLB2O/SLB2T) **[29,30]:** With this term we refer to binary tournament selection. Two individuals are chosen randomly and finally, the individual with the higher fitness value is selected. The process is repeated for generating a second parent individual address in the case of SLB2T.

3.3.4 Remaining instructions

Apart from the genetic operators, instructions for invoking fitness evaluation, configuring the termination criterion of a GA, defining offspring survival strategies and for handling pseudo-random number generation are needed as well.

Fitness evaluations can be specified either in parallel for two parent individuals or for one of the parent chromosomes only. Generally, fitness instructions are entirely application-specific which is the reason for not predefining any such instruction in VGP-I, but rather dedicating a complete set of secondary opcodes ($func$ in L-format) to add the desired instructions.

GA termination instructions tend to be more eager for reuse. Two basic forms of termination criteria have been defined in VGP-I. TCNCMAX is used when a good (or optimal) solution is known for a given problem. Then, when an individual ranks equal or better fitness to the best possible fitness, the VGP-I processor should halt. The TCNNGEN form is used for terminating the process after a number of N executions of the termination criterion instruction.

In the VGP-I architecture, the default behavior for replacing individuals is that parents are always replaced by their corresponding offspring in place. This is implemented intrinsically for each reproduction operator instruction (crossover or mutation) unless it is overriden by the PCSR. Survival-driven replacement strategies are supported by SURV instructions that can be used for single-chromosome individuals. Survival operations decide on the utilization of intermediate values (new chromosomes with their fitness data and addresses). Currently, only the SVRIB instruction has been specified, according to which an offspring replaces its parent in case of higher fitness.

The last non-trivial instructions in VGP-I are LDEXT, LDRNG, SETRNGR and EVALP. A population can be initialized to specific values by successive uses of the LDEXT instruction or to random chromosome values by LDRNG. SETRNGR sets the resolution for crossover and mutation operator probabilities, which are evaluated by using the EVALP instruction.

As a suggestive instruction subset not defined in the fixed VGP-I ISA, instructions LBCRB, LBFRB, LBS are used for storing immediate values in any architectural register. The fixed VGP-I ISA makes provision for direct access only to the CRB (via LDEXT). Since VGP-I microarchitectures are most likely considered to be integrated as soft cores in FPGA systems-on-chip (SoCs), we can take advantage of bitstream initialization and partial reconfiguration capabilities of FPGAs.

3.4 The VGP-I register architecture

Table 3 details the proposed register set for the VGP-I architecture. The first four registers (*PSIZE, NCHROM, CLEN, ASIZE*) along with the operator probability registers (*CPROB, MPROB, PRNGR*) provide the algorithmic and implementation parameters for a GA. The actual architectural registers are: *FCRA, MCRA* holding individual addresses calculated by selection and the *BFITx* registers used by the fitness and termination criterion primitives. The *PCSR* is the processor control/status register for keeping the necessary flags for the processor. Finally, the *TxCRD* and *TFITx* registers are only used for explicit survival mechanisms.

Table 3
The VGP-I register architecture.

Resource	Usage
[C|F]RB[·]	Chromosome/Fitness register file
PSIZE	Population size
NCHROM	Number of chromosomes per individual
CLEN	Chromosome length
ASIZE	Allele size
[F|M]CRA	Parent 1/2 (father/mother) chromosome address
BFIT[D|A]	Best fitness ever register data/address
BFITP	Best fitness possible
PRNGR	PRNG resolution
[C|M]PROB	Crossover/mutation probability
PCSR	Processor control/status register
	bit 0: Halt assertion flag
	bit 1/2: Enable the use of [C|M]PROB
	bit 3: Explicit survival mechanisms
	bit 4: On-the-fly allele size adaptation
T[S|D]CRD	Temporary offspring 1/2 (son/daughter) chromosome
TFIT[D|A]	Temporary fitness data/address

3.5 Conceptual block diagram of a VGP-I processor

A block diagram of an architecture complying to these principles can be seen in Fig. 2. This figure exposes a pipelined execution model for a VGP-I architecture, which can be justified if the greatest portion in typical execution profiles is consumed by single-cycle instructions. Notably, the specific mechanisms implementing fitness and termination criterion evaluation are not considered part of the VGP-I architecture. Instead, only the appropriate interfacing to these units is specified in terms of the BFITD, BFITA, BFITP and PCSR architectural registers. Naturally, for many problems, a locally or globally optimal fitness is not known and in these cases BFITP is unused.

4 Hardware realizations of VGP-I subsets

4.1 A prototype processor implementation for a VGP-I subset

Following the architectural scheme of Fig. 2 we have implemented a prototype microarchitecture in VHDL, named SVGP-I, for a VGP-I subset. SVGP-I employs a single execution stage, it is thus organized as a 3-stage pipeline. SVGP-I has been used for targeting simple case studies, as the well-known

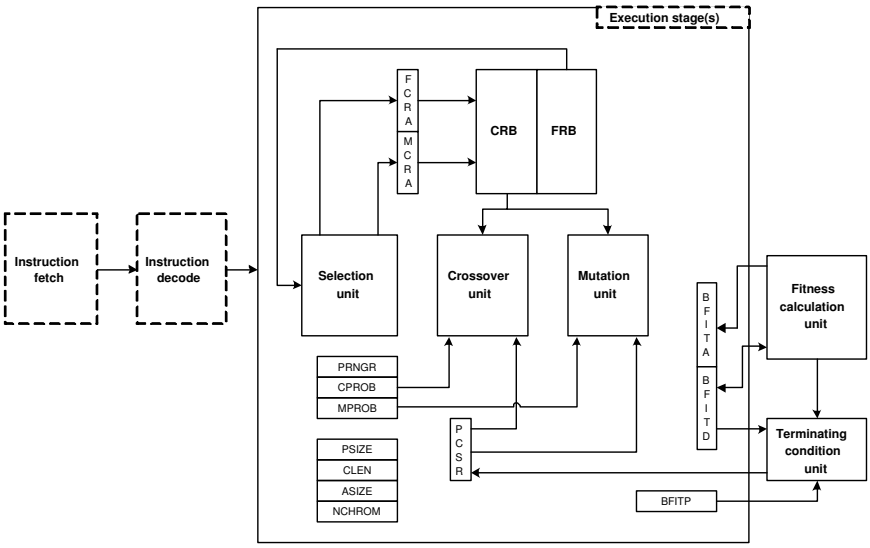


Fig. 2. Overview diagram of a typical VGP-I architecture.

onemax problem and a string-matching problem named *stringmat*. The SVGP-I supports population sizes up to 16 consisting of 16-bit single-chromosome individuals.

The fitness calculation and termination criterion unit (FTCU) for the specific problems is shown in Fig. 3. The 'genconst' register contains the expression to be matched. When a chromosome value (parent 1 and parent 2 inputs) is compared to the value 0xFFFF, the SVGP-I solves *onemax*, and when an arbitrary value is used for comparison, the same hardware is used for *stringmat*. When a fitness instruction is applied, fitness values are determined by a 'population count' operation performed by the 'one's counter' components that sum up the number of 1's in a given chromosome. The produced values are then available as parent 1/2 fitnesses to be written to the corresponding FRB addresses. They are also applied to a magnitude comparator (GT1) to determine the relative winner. This value is then compared to the BFITD register and if better the BFITA and BFITD registers are updated. A termination criterion instruction uses EQ3 to compare this value to the best possible fitness (BFITP) to determine if the termination criterion is met. If this is the case, bit 0 of the PCSR is updated and a halt condition is issued.

4.1.1 Crossover datapath

Fig. 4 shows the SVGP-I datapath for the crossover types detailed in paragraph 3.3.1. In its general form, this component is characterized by two parameters: n (chromosome bitwidth) and k (minimum allele size) and any specific implementation can thus be denoted as (n,k). In the given figure, a (16,4) crossover unit is shown, however an (n,1) crossover unit has also been implemented. The CRMN type however can only be used for allele sizes equal to a

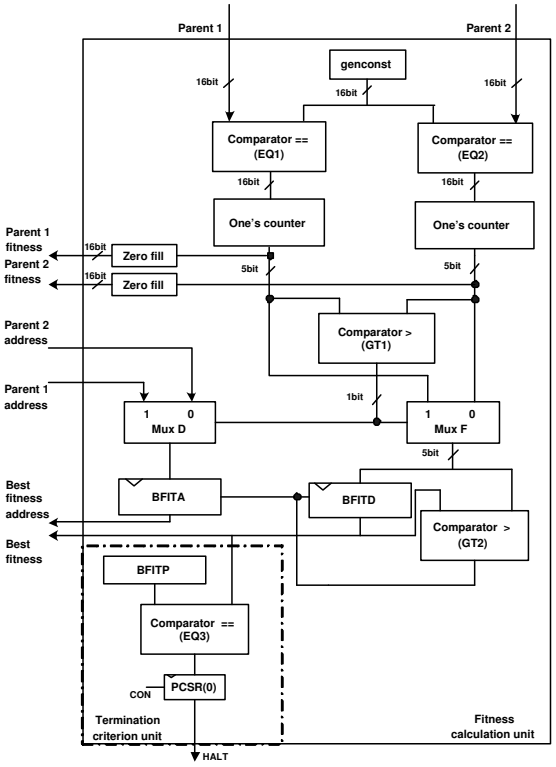


Fig. 3. Problem-specific fitness evaluation and terminating criterion unit (FTCU).

power-of-2.

The crossover datapath, uses a form of micro-SIMD [7] in order to execute mean crossover. This technique enables concurrent processing of a number of parent alleles in the same clock cycle. For that reason, SIMD-aware adders, subtractors and shifters are used. The actual crossover procedure takes place at the bottom multiplexer network. In the shown case, inputs '0', '1' are connected to the unmodified parent 1, 2 alleles for both offspring. For producing offspring 1, the '2','3' inputs are wired to $(F_a + M_a)/2$ and $(F_a + M_a - 1)/2$, respectively, with F_a,M_a denoting parent 1,2 alleles at the same position. The corresponding inputs are connected to $(F_a + M_a - 1)/2$ and $(F_a + M_a)/2$ and are selected for producing offspring 2 of mean crossover.

4.1.2 Mutation datapath

The mutation unit of SVGP-I has been implemented as the datapath shown in Fig. 5. An offset value is added or subtracted by each parent allele and the appropriate result is controlled by signal *dir*. An unmodified parent allele can be passed through for a zero-valued offset. Thus, at the output of the 'MuxB' multiplexer, $F_a \pm offset$ is available. The bottom multiplexer network is comprised of n/k multiplexers (n, k defined in Fig. 4) that either copy the produced allele value of $Fa \pm offset$ (input '0'), a zero value (input '1') or

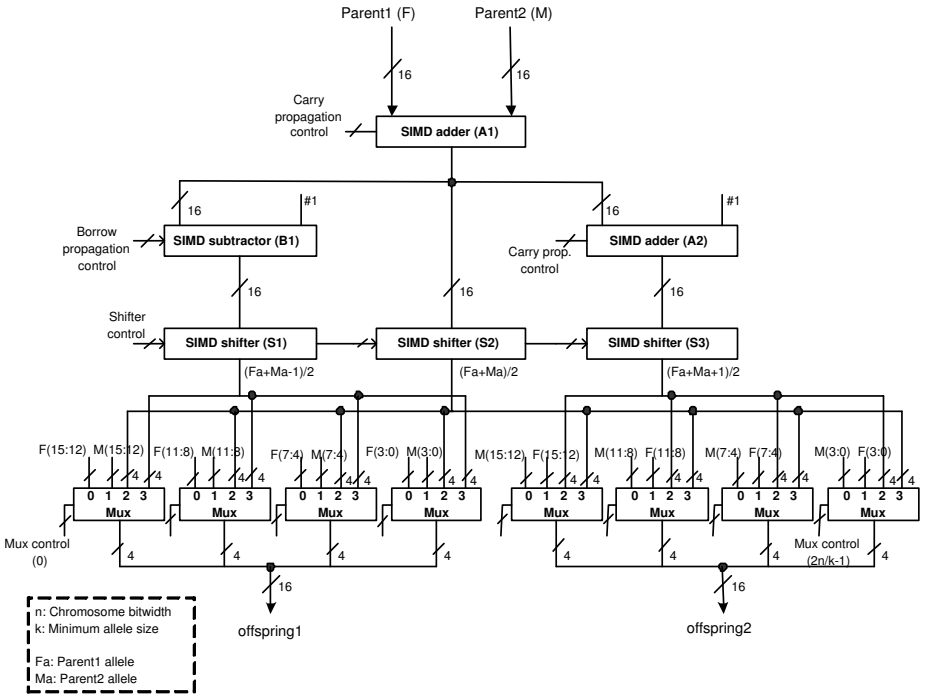


Fig. 4. Crossover unit datapath.

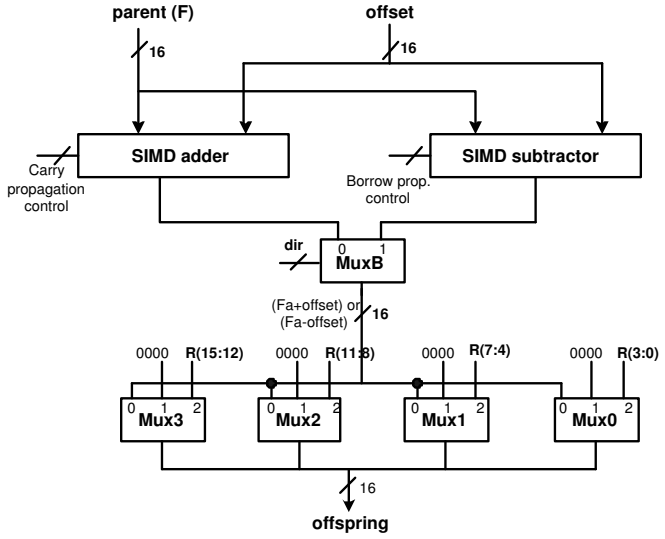


Fig. 5. Mutation unit datapath.

a random allele (R_a) generated by an LFSR ('2'). For mutating both parent chromosomes at the same time, a second instance of the shown hardware can be used.

4.1.3 Selection datapath

The selection datapath is depicted in Fig. 6. Initially, the reset signal sets the contents of the parent address registers to zero. When the SLRNO type is executed, $rds = 1$, $wadr1 = 1$ and $wadr2 = 0$, and by the end of the clock cycle FCRA is written with the random value of LFSR1. In case that the SLRNT type is executed, $rds = 1$, $wadr1 = 1$ and $wadr2 = 1$, so that the LFSR1 and LFSR2 random values will be stored to FCRA and MCRA, respectively.

SLB2O also takes a single clock cycle to be executed. The control signals should be $rds = 0$, $init = 0$, and $EPI = 1$, and the FCRA is determined to be the address with the greater fitness value among the LFSR1 and LFSR2 values. SLB2T is completed in two clock cycles. During the first clock cycle, control values are as for SLB2O, and at the second clock cycle $init = 1$ to prevent LFSR random values from being equal to the chosen parent 1 address from the first clock cycle. The effect of SLB2T is to update the FCRA and MCRA registers, which hold the parent 1 and 2 address, respectively.

The LD and FL control signals are used in the initialization process of the VGP-I prototype. While given chromosome values are being stored into the destination register addresses $LCOUNT1$ and $LCOUNT2$ of the CRB register file, LD is set high. $LD = 1$ ensures that the chromosome values will be stored into the right positions of CRB and that these positions can be controlled by determining $LCOUNT1$ and $LCOUNT2$ values. During normal operation in a GA kernel, it is necessary to compute the fitness value of every individual in the population. During this process, FL should be set high. $LCOUNT1$ and $LCOUNT2$ addresses provide that the individual's fitness will be stored into FRB in the same positions those individuals have in the CRB.

4.1.4 Implementation details for the specialized register files

As imposed by the semantics of crossover and selection, concurrent access to either 2 read or 2 write ports is highly desirable, to be able to concurrently process 2 parent individuals or fitness values, respectively. Most RAM generators for ASIC processes do not provide memory macros with 2 write ports, in contrast to recent FPGA architectures. It is possible to utilize dual-port block RAMs such as the *RAMB16_Sm_Sn* component in Xilinx Spartan-3 FPGAs [31] for implementing the CRB and FRB register files of the VGP-I architecture. The corresponding read/write addresses can be shared (they are connected to FCRA and MCRA) thus only 2 address ports are needed. This situation is ideal for using the particular block RAM architecture to perfectly fit the GA domain requirements.

In Fig. 7 the design of the actual register file template used for both CRB

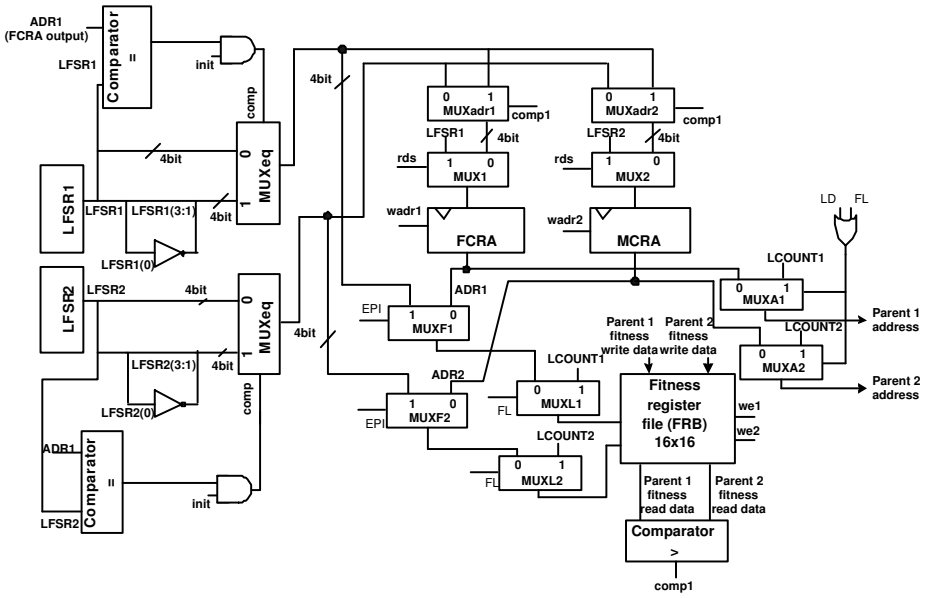


Fig. 6. Datapath for selection operations.

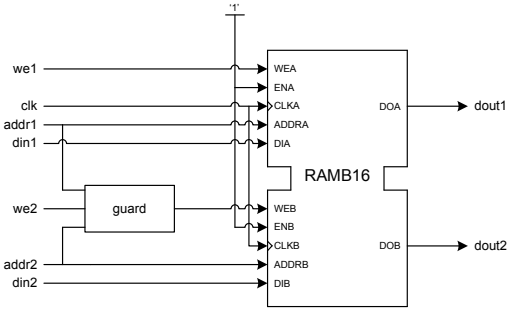


Fig. 7. The VGP-I register file module implemented with Xilinx block RAM (RAMB16_S18_S18) under the cell configuration 1024×16.

Table 4
Synthesis report for the register file template.

Device	Spartan3 xc3s200-4-pq208
Slices	3 out of 1920 (0%)
BRAMs	1 (BRAM16_S36_S36)
Max clock frequency	143.3 MHz

and FRB is shown. The capability of dual-port simultaneous writes is enabled by the appropriate local control logic, which compares the offspring addresses and in case these are equal, the write port addressed by MCRA is prevented from writing back to the chromosome register file.

The design style for dual-port block RAM with two write ports is the one suggested by the vendor [32] with the memory array declared as a VHDL-93 shared variable. The corresponding synthesis report is shown in Table 4.

Table 5
Logic synthesis results for an ASIC and FPGA implementation of SVGP-I.

Design	ASIC (0.13μm)	FPGA (XC3S200)
Crossover datapath	480 gates / 2.07ns	81 slices (out of 1920) / –
Mutation datapath	214 gates / 1.28ns	41 slices / –
Selection datapath	114 gates / 0.28ns	15 slices / 4.89ns
FCU,TCU	–	42 slices / 6.03ns
SVGP-I without FCU,TCU	6160 gates / 406.8MHz	–
SVGP-I with FCU,TCU	6261 gates / 394.5MHz	320 slices / 57.6MHz

It is possible to map the entire register set (38 in SVGP-I) on a single block RAM, given that an index prefix unit (IPU) is added in the microarchitecture and is utilized during instruction decode. This has been done for a VGP-I instruction-accurate ArchC model [33].

4.2 Hardware characterization of the SVGP-I prototype

The SVGP-I prototype has been synthesized on a popular 0.13μm ASIC process using LeonardoSpectrum and on a Xilinx Spartan-3 XC3S200 FPGA with Xilinx ISE 7.1i (Webpack). It should be noted that for the XC3S200 version all architectural registers are implemented in block RAMs. The synthesis results are summarized in Table 5.

4.3 Instruction set extensions for the Nios II processor based on a VGP-I subset

In addition to the SVGP-I prototype, we have designed custom instruction units for GA instruction set extension of the Nios II processor soft-core. The crossover custom instruction unit only supports the CRCSP, CRCDP and CRMX types, it is however capable of processing alleles of different sizes (1 to 32 bits). Crossover generates two output results while Nios II is restricted to committing a single result per instruction. For this reason, a data transfer unit implementing move instructions among the general-purpose and the internal (custom) register file of Nios II has been designed. These instructions are essentially the same to MOVI2S and MOVS2I of the DLX instruction set. Thus, for committing the second offspring chromosome, a MOVS2I is executed following the crossover instruction. The mutation custom instruction unit supports MTSPD, MTSPR and MTAP for any allele size up to 32 bits. The selection operations are implemented on Nios II in the form of short subroutines including custom instructions for smaller basic operations accessed

```
struct individual *population;
unsigned int address1,address2,FCRA,MCRA;
int fitness1,fitness2;
⋮
address1 = PRNG(5);
address2 = PRNG(5);
fitness1 = PCOUNT(population[address1]);
fitness2 = PCOUNT(population[address2]);
FCRA = SELECT_FIRST(fitness1,fitness2);
address1 = PRNG(5);
address2 = PRNG(5);
fitness1 = PCOUNT(population[address1]);
fitness2 = PCOUNT(population[address2]);
MCRA = SELECT_SECOND(fitness1,fitness2);
```

Fig. 8
C code for performing the SLB2T selection type on Nios II.

for selection. The actual hardware operators are: PRNG which generates an n-bit random integer, PCOUNT to perform the actual population-count operation used for fitness evaluation in some application examples (as *onemax*), and SELECT_FIRST, SELECT_SECOND to produce new addresses for individuals in the population. Fig. 8 shows an implementation of the SLB2T selection type with the assistance of custom instruction extensions.

Further, we have assessed the cost of incorporating the GA instruction set extensions to Nios II. The custom instructions were added to the Nios II/s version of the core and a SoPC (system-on-programmable-chip) was built with the Nios II 5.1 Development Kit. The default system organization requires a hardware cost of 3672 ALUTs (8% of the device capacity), 2033 registers and 8 9-bit DSP elements on the EP2S60F672C5 Stratix-II FPGA device. The GA extensions demand 498 ALUTs and 198 registers more, which is a rather moderate cost. The maximum clock frequency was reduced by 10.2% (from 83.67 MHz to 75.15 MHz) which would be acceptable if it is justified by much increased cycle performance.

5 Demonstration applications and performance results

In this section, the performance of VGP-I architectures is evaluated in terms of the efficiency of the genetic operators (subsection 5.1) as well as on standalone GAs written for the VGP-I. As demonstration applications, we have chosen a steady-state GA (subsection 5.2) and a TSP solver (subsection 5.3).

Table 6
Performance comparison of genetic operators on an embedded RISC and a VGP-I architecture model.

Operator	MIPS-I instr.	VGP-I instr.	VGP-I cycles	Nios II custom instr. cycles
CRCSP	63	1	1	2
CRCDP	114	1	1	2
CRMX	75	1	1	2
CRAMX	56	1	n_a+1	-
CRMN	61	1	1	-
MTSPD	95	1	1	2
MTSPR	90	1	1	2
MTMP	167	1	n_a+1	-
MTAP	128	1	n_a+1	2
SLRNO	53	1	1	2
SLRNT	86	1	1	3
SLB2O	136	1	1	7
SLB2T	198	1	2	14

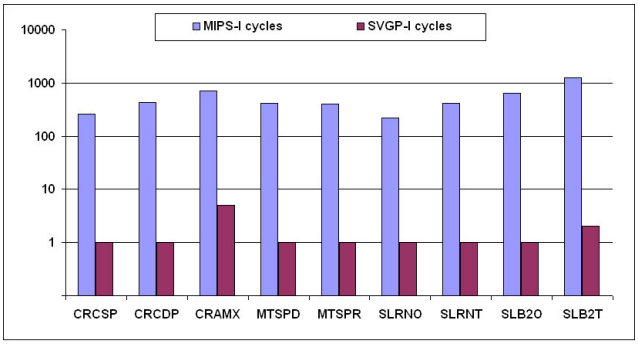
5.1 Performance comparisons of VGP-I architectures against general-purpose embedded processors on the implementation of genetic operators

In order to evaluate the performance benefits of VGP-I over a general-purpose software implementation of GAs, we have developed a C library of genetic operators and problems, named *gentest*. *gentest* implements static versions of some genetic operators found in GAUL with much reduced overhead in data structure maintaining. Although a compiler for VGP-I is not available, it is reasonable to assume that the genetic operators can be instantiated by compiler-known functions, similar to our Nios II approach.

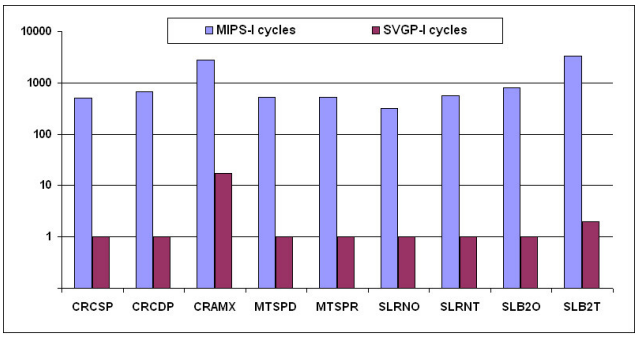
Table 6 summarizes typical expected performance for a MIPS-like and a VGP-I processor, while Fig. 9 gives actual cycle measurements for MIPS R3000 and SVGP-I over a set of genetic operators. The performance improvement is typically 2 to 3 orders of magnitude for any operator and allele size/number of alleles combination.

5.2 Demonstration application 1: A typical GA for VGP-I

For demonstrating the flexibility of the VGP-I architecture, a steady-state GA has been coded in the native VGP-I assembly for solving the *onemax* problem and a number of closely-related problems. A reference C implementation of the algorithm with compiler intrinsic functions that would directly map to the corresponding assembly instructions is shown in Fig. 10. Literals in capitals,



(a) $n_a = 4, al_s = 4$



(b) $n_a = 1, al_s = 16$

Fig. 9. Cycle performance for SVGP-I against software running on a MIPS-like processor.

if not mapped to storage resources of Table 3, denote integer-valued compile-time constants. *gl* is an input variable determining the allele length, while a specific operator is specified as x_TYPE (where x is one of the genetic operator instruction mnemonics of Table 2) applied via corresponding wrapping functions.

The VGP-I performance for *onemax* was compared to a software implementation on a MIPS-I processor model written for the *SGA-C* package [34]. The algorithmic parameters for the two contrasted realizations are given in Table 7. For the 16-bit VGP-I ISA, the corresponding cycle-accurate model was written in ArchC. For the MIPS-I simulations, the R3000 model from [33] was used. The C code for *onemax* was compiled with *gcc* (3.3.1) for the options *-O3 -mips1 -msoft-float*. As an RNG, we have ported a C implementation of a Mersenne Twister [35] to the ArchC models.

In addition to *onemax*, a few more applications were composed according to the SSGA. *stringmat* is a simple string-matching problem, while *stringmat-sm* is a variation with a smoother fitness function. In *altzo*, a pattern of alternating ones and zeroes should be matched, and *maze2d* is a solver for 16×16 2-dimensional maze. The *onemax*, *altzo* and *stringmat* problems have been implemented on the same FCU of the SVGP-I microarchitecture, as well. For each given problem, only the fitness function is altered by employing a differ-

```
unsigned int CRB[16],FRB[16];
⋮

InitializeParameters(CLEN,PSIZE,PRNGR,CPROB,MPROB,BFITP);
InitializePopulation(CRB,gl);
FitnessEvaluation(CRB,FRB,gl);

while (!get_bits(PCSR,0,0)) {
 SelectTwoIndividuals(SELECTION_TYPE);
 EvaluateProbability(CROSSOVER);
 Crossover(CROSSOVER_TYPE,gl);
 FitnessOfBothParents(gl);
 TerminationCriterion(TCNCMAX);
 EvaluateProbability(MUTATION);
 Mutation(MUTATION_TYPE,gl);
 FitnessOfParent1(gl);
 TerminationCriterion(TCNCMAX); }
```

Fig. 10
Generic C implementation of a steady-state GA to be compiled to VGP-I assembly.

Table 7
Algorithmic parameters for the *onemax* problem.

Description	On VGP-I	On MIPS-I
Population size	if (n<16) then n else 16	n
Chromosome length	n	n
Allele size	1	1
Crossover probability	0.7	0.7
Mutation probability	0.1	0.1

ent $lc1$ value. As the termination criterion instruction, the TCNCMAX was used, since for all problems an optimal solution was known a priori.

The performance results of Table 8 have been collected for *onemax* and are typical of this application set. Several crossover and mutation types have been examined on GA runs on the VGP-I, while for the MIPS-I version we have used the single-point crossover and single-point randomize mutation types that are already specified in SGA-C. It has been found that CRMN and MTAP dramatically accelerate VGP-I performance on *onemax*, providing rapid convergence to optimum solutions as can be seen in Table 8. Other crossover types as allele mixing and n-point crossover with n=2-4 do not provide fast convergence with reasonable quality. It has also been concluded that the binary tournament selection is far superior than random selection as expected. To study this effect we have performed further simulations on both the VGP-I

Table 8
Performance results for the *onemax* problem (VGP-I vs MIPS-I).

Vector length	Crossover/Mutation	VGP-I cycles	MIPS-I cycles	Speedup
8	CRCDP/MTSPR	24547	603657	24.6
12	CRCDP/MTSPR	116863	1778376	15.2
16	CRCDP/MTSPR	759690	4810022	6.3
24	CRCDP/MTSPR	684423	17885438	26.1
32	CRCDP/MTSPR	1868996	214813037	114.9
24	CRMN/MTSPR	114	–	–
24	CRCDP/MTAP	155	–	–
32	CRMN/MTSPR	93	–	–
32	CRCDP/MTAP	558	–	–

and MIPS-I models with different selection operators while retaining the remaining parameters of Table 7 and with crossover and mutation types fixed to CRCDP and MTSPR, respectively. By examining the results, we can see that when random selection is applied, execution time is increased by a factor of about 7.6× against the case of binary tournament selection. This result is not significantly affected by the SVRIB replacement strategy. The performance benefits of tournament against random selection are more profound if:

- Tournament size is larger. For a tournament size of 8 on a population of 16, an additional speedup of about 10× is attained compared to binary tournament.
- A form of elitism is applied for maintaining good solutions in the population.

Further, it is not clear if performing additional selection prior to mutation has any benefits. Notably, the well-known generational SGA [1] reuses the result of the first selection (SLB2T in our case).

The SGA version of the *onemax* GA solver was used as a benchmark for the Nios II case. The achieved speedups for the examined chromosome lengths (16, 24, 32 bits) range from 14× to 22×, gradually improving for larger bitstring lengths. The performance results are shown in Table 9, averaged over five different runs.

5.3 Demonstration application 2: A TSP solver

The TSP (Traveling Salesman Problem) is a classic path-planning optimization problem that is known to be NP-hard. Efficient methods to solve the TSP exist (CONCORDE [36] and other variations of the Kernighan-Lin heuristic), but the purpose of this subsection is to exhibit the programmability of VGP-I processors by describing a simple TSP GA for VGP-I. The TSP GA differs

Table 9
Performance results for *onemax* on Nios II. A software implementation (SW) is contrasted against Nios II with custom instructions (HW) using the same SGA algorithm.

Length	GA operators in HW	Fitness	Cycles	Exec. time	Speedup
16	CRCDP/MTSPR/SLRNT	HW	1571093	20.91ms	14.96
16	CRCDP/MTSPR/SLB2T	HW	1632187	21.72ms	14.40
16	CRCDP/MTSPR/SLB2T	SW	3224751	42.91ms	7.29
16	None	SW	26165315	312.73ms	1.0
24	CRCDP/MTSPR/SLRNT	HW	3988528	53.07ms	21.31
24	CRCDP/MTSPR/SLB2T	HW	4149412	55.21ms	20.48
24	CRCDP/MTSPR/SLB2T	SW	10528910	140.10ms	8.07
24	None	SW	94613612	1.130s	1.0
32	CRCDP/MTSPR/SLRNT	HW	36388037	484.18ms	22.65
32	CRCDP/MTSPR/SLB2T	HW	37802983	503.01ms	21.80
32	CRCDP/MTSPR/SLB2T	SW	114536158	1.524s	7.19
32	None	SW	917404036	10.965 s	1.0

significantly from demonstration application 1. Mixing crossover is used in the forced exchange mode (EXNG instruction) to provide a form of swap mutation. The entire population comprises of a single individual with its alleles distributed over corresponding CRB registers. Such flexibility is not available in hardware GAs, and it is a differentiating feature of VGP-I. In this GA, the task of selection is to choose two alleles situated in different CRB registers for exchange. We have used random selection (SLRNT) since it does not require fitness evaluation.

The actual GA implementation on the VGP-I is shown in Fig. 11. The MVI2x instructions are macro-instructions for loading the upper and lower halfwords of an architectural register by immediates. For comparison purposes, we have imported a compact version of the GALOPPS [37] implementation for the TSP to SGA-C [34]. *tsptest8, tsptest12, tsptest16* are synthetic TSP data instances while *ulysses22, dantzig42, berlin52* have been obtained from TSPLIB [38]. The reference C implementation was compiled and executed on a MIPS-I model under the same procedure as in the previous subsection, and the results were averaged for a series of five runs. It should be noted that the reference TSP implementation was based on a generational algorithm. For the SGA-C algorithm, the same values were used for the crossover and mutation probabilities as for *onemax* while $NCHROM$ was set equal to the number of cities (*ncities*).

The performance results for the TSP problem are shown in Table 10. Our simplistic GA on the VGP-I can be used under the assumption that slightly larger path lengths can be accepted. Thus, the program in Fig. 11 can be used for environments were a quality-time metric tradeoff is meaningful, e.g. trajec-

```
TSPSolver
   // population-size=1, number-of-chromosomes=ncities
   // allele-size=log2(ncities), gl=0
                  ⋮
   MVI2CRB        C0, 0
                  ⋮
   MVI2CRB        Cncities−1, ncities-1
   MVI2FCRA       0
   FITTSPF        gl
tsp_loop:
   SLRNT
   EXNG
   FITTSPF        gl
   SVRIB
   TCNNGEN
   JMP            tsp_loop
   HALT
```

Fig. 11
VGP-I assembly program for the TSP solver.

Table 10
Performance results for the TSP.

TSP data	VGP-I cycles	Known optimal tour	% path length diff. (VGP-I/optimal tour)
tsptest8	4701	132	1.36
tsptest12	18645	132	2.38
tsptest16	68503	230	5.48
ulysses22	224804	2030	3.88
dantzig42	550667	675	4.74
berlin52	4612867	7542	44.65

tory planning with strict real-time constraints. Due to the small population size (only a single individual is used) and the unavailability of local search mechanisms, the diversion of the maximum fitness values on the VGP-I to the fitness of known optimal tours is significantly increased for a number of cities larger than approximately 50.

6 Conclusions

In this paper, a specification for the design of programmable processors for the execution of genetic algorithms has been presented. The proposed architecture, named VGP-I, provides flexibility through a specialized instruction

set with multiple types of genetic operators realized on custom functional units, and is optimized for implementation on contemporary FPGA devices. To evaluate our concept we have developed both an ASIP and a processor extension realization of large VGP-I subsets. The ASIP prototype has been used on a number of GA problems delivering significant performance boost over other approaches involving programmable engines. In addition, a Nios II soft-core processor with GA instruction set extensions provides application speedups of around 20×. Future work regards the introduction of additional survival mechanisms to the VGP architecture and the development of SoC reference designs with a VGP-I component as a coprocessor to a general-purpose embedded processor, accessed through either a local or the main on-chip bus.

References

[1] D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, MA, USA, 1989.

[2] J. H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, Michigan, USA, 1975.

[3] D. Burns, K. May, T. Renz, V. Ross, Spiraling in on speed-ups of genetic algorithm solvers for coupled non-linear ODE system parameterization and DNA code word library synthesis, in: 2005 Military and Aerospace PLD International Conference (MAPLD 2005), Washington, DC, USA, 2005.

[4] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[5] P. Martin, A hardware implementation of a genetic programming system using FPGAs and Handel-C, Genetic Programming and Evolvable Machines 2 (4) (2001) 317–343.

[6] S. Koizumi, S. Wakabayashi, T. Koide, K. Fujiwara, N. Imura, A RISC processor for high-speed execution of genetic algorithms, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), San Francisco, California, USA, 2001, pp. 1338–1345.

[7] R. B. Lee, Multimedia extensions for general-purpose processors, in: Proceedings of the IEEE International Workshop on Signal Processing Systems, Leicester, United Kingdom, 1997, pp. 9–23.

[8] G. M. Megson, I. M. Bland, Synthesis of a systolic array genetic algorithm, in: Proceedings of the 12th International Parallel Processing Symposium, Washington, DC, USA, 1998, pp. 316–320.

[9] Altera Nios II home page.
URL `http://www.altera.com/products/ip/processors/nios2/`

[10] J.-P. Choi, K.-R. Han, H.-J. Song, I.-J. Hwang, G.-Y. Song, Design of a microprocessor with genetic instructions, in: Proceedings of the 2002 International Technical Conference on Circuits / Systems, Computers and Communications (ITC-CSCC 2002), Phuket, Thailand, 2002, pp. 666–669.

[11] S. D. Scott, A. Samal, S. Seth, HGA: A hardware-based genetic algorithm, in: Proceedings of the ACM/SIGDA 3rd International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 1995, pp. 53–59.

[12] P. Graham, B. Nelson, Genetic algorithms in software and in hardware - a performance analysis of workstation and custom computing machine implementations, in: IEEE Symposium on Field-Programmable Custom Computing Machines, Los Alamitos, CA, USA, 1996, pp. 216–225.

[13] O. Kitaura, H. Asada, M. Matsuzaki, T. Kawai, H. Ando, T. Shimada, A custom computing machine for genetic algorithms without pipeline stalls, in: Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics, Vol. V, Tokyo, Japan, 1999, pp. 577–583.

[14] M. K. Pakhira, R. K. De, A hardware pipeline for function optimization using genetic algorithms, in: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO'05), Washington, DC, USA, 2005, pp. 949–956.

[15] M. Tommiska, J. Vuori, Implementation of genetic algorithms with programmable logic devices, in: Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA), Vaasa, Finland, 1996, pp. 71–78.

[16] G. Koonar, S. Areibi, M. Moussa, Hardware implementation of genetic algorithms for VLSI CAD design, in: ISCA Int. Conf. on Computer Applications in Industry and Engineering, San Diego, CA, USA, 2002.

[17] B. Shackleford, G. Snider, R. J. Carter, E. Okushi, M. Yasuda, K. Seo, H. Yasuura, A high-performance, pipelined, FPGA-based genetic algorithm module, Genetic Programming and Evolvable Machines 2 (1) (2001) 33–60.

[18] S. Wakabayashi, T. Koide, N. Toshine, M. Goto, Y. Nakayama, K. Hatta, An LSI implementation of an adaptive genetic algorithm with on-the-fly crossover operator selection, in: Proceedings of the Asia and South Pacific Design Automation Conference, Vol. 1, Wanchai, Hong Kong, 1999, pp. 37–40.

[19] W. M. Spears, Adapting crossover in evolutionary algorithms, in: Proceedings of the Evolutionary Programming Conference, 1995, pp. 367–384.

[20] K. A. D. Jong, An analysis of the behavior of a class of genetic adaptive systems, Ph.D. thesis, Department of Computer and Communication Sciences, University of Michigan (1975).

[21] G. Syswerda, Uniform crossover in genetic algorithms, in: Proceedings of the 3rd International Conference on Genetic Algorithms, Fairfax, VA, USA, 1989, pp. 2–9.

[22] ARM Ltd.
URL http://www.arm.com

[23] MIPS technologies Inc.
URL http://www.mips.com

[24] Gaisler research.
URL http://www.gaisler.com

[25] Genetic algorithm utility library.
URL http://gaul.sourceforge.net

[26] G. Syswerda, Schedule optimization using genetic algorithms, in: L. Davis (Ed.), Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, NY, USA, 1991, pp. 332–349.

[27] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, 3rd Edition, Springer, Berlin, Germany, 1996.

[28] LFSR testbench utility.
URL http://www.jnicolle.com/LFSR/

[29] A. Brindle, Genetic algorithms for function optimization, Ph.D. thesis, University of Alberta, Edmonton, Canada (1981).

[30] D. E. Goldberg, K. Deb, A comparative analysis of selection schemes used in genetic algorithms, in: G. J. E. Rawlins (Ed.), Foundations of Genetic Algorithms, Morgan Kaufmann, San Mateo, CA, USA, 1991, pp. 69–93.

[31] Xilinx home page.
URL http://www.xilinx.com

[32] Xilinx ISE 7.1i (Webpack), Xilinx Inc., XST User Guide.

[33] The ArchC resource center.
URL http://www.archc.org

[34] R. E. Smith, D. E. Goldberg, J. A. Earickson, SGA-C: A C-language implementation of a simple genetic algorithm, TCGA Report No. 91002, The University of Alabama, Tuscaloosa, AL, USA (March 1994).

[35] M. Matsumoto, N. Nishimura, Mersenne Twister pseudorandom number generator in C.
URL http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

[36] CONCORDE.
URL http://www.tsp.gatech.edu/concorde.html

[37] GALOPPS.
URL http://garage.cps.msu.edu/software/galopps/

[38] TSPLIB.
URL
http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/