
An Evolvable Hardware Platform based on DSP and FPTA

M. I. Ferguson*, A. Stoica, D. Keymeulen, R. Zebulum, V. Duong

*fergie@jpl.nasa.gov

Center for Integrated Space Microsystems

Jet Propulsion Laboratory

California Institute of Technology

Pasadena CA 91109, USA

Abstract

The field of evolvable hardware has been facing a consistent problem since its inception, that the time taken to find an evolved solution is often limited because of time taken in the simulation of the hardware. We describe in this paper the rationale and design of an evolvable hardware (EHW) platform based on the use of a stand-alone processor (DSP) and a Field Programmable Transistor Array (FPTA). We demonstrate an improvement of 3 to 4 orders of magnitude over state-of-the-art simulation techniques implemented on a super-SPARC processor.

1 Introduction

The field of evolvable hardware (EHW) has been facing a consistent problem since its inception; that the time taken to find a solution often imposes limits on the size of the final circuit because the evaluation time is too great. This stems from the fact that until recently all algorithms have been running simulations of the hardware, rather than testing on hardware directly. The JPL EHW research team has made an attempt to mitigate this by developing an integrated circuit called a field programmable transistor array (FPTA). Evaluating candidate solutions on this hardware vastly reduces the evaluation time by removing the circuit simulation time. In order to efficiently interact with this new device, a dedicated processor is provided to both perform the evolutionary algorithm, evaluate the performance and perform the analog I/O with the chip.

This document provides technical rationale and explanation for the design of the evolutionary algorithm implemented for the stand-alone DSP platform. The first section will provide an overview of the system and the system architecture to provide context for the more detailed explanations to follow. The next section will discuss the implementation of the genetic algorithm. The following sections will in turn discuss the downloading of individuals to the FPTA and the methodology behind the stimulus/response cycle.

1.1 Related Work

Several other projects related to evolutionary electronics have been documented in the literature, most notably is the work by

Several systems have been developed to perform Genetic Algorithms (GAs) on FPGAs.

We know of no other group doing research into the area of evolutionary electronics using a stand-alone processor and an evolutionary hardware platform.

2 System Overview

The goal of the system is to discover a configuration for the FPTA to perform a function specified as an input to the algorithm. This goal is achieved with the use of a genetic algorithm, which operates on a population of individual configuration candidates. The term 'chromosome' is used to represent one of the configuration bit-stream candidates and a population is made up of individual chromosomes. The GA operates by testing each of the chromosomes, also called individuals, on the FPTA and

evaluating its performance or fitness. When all individuals have been evaluated a new population is formed by using genetic operators on the old population. Genetic operators include such functions as crossover and mutation. This process of generating and evaluating populations continues until a solution is found or it is determined that the search is not likely to converge to a solution.

2.1 System Architecture

The system consists of a PC connected through JTAG to an Innovative Integration SBC67 stand-alone DSP board[2] which is then connected to the FPTA designed at JPL[1]. The SBC67 has a Texas Instruments (TI) TMS320C6701 floating-point processor with 128KB internal SRAM and 16MB of external SRAM. It also has two add-on modules connected through a proprietary OMNIBUS interface, a *SERVO-16* providing 16 analog input and output with 16-bits of precision and simultaneous sampling at 100kSample/sec and a *DIG-32* for 32 additional digital I/O operating at 7.5Mhz.[3]

The evolutionary algorithms are developed on the PC using the TI Code Composer environment. The compiled code is then downloaded to the DSP via a JTAG connection. The JTAG connection is used subsequently to monitor and control the algorithm pending development of a host application which can communicate more efficiently with the DSP via USB¹.

The evolutionary algorithm is executed on the DSP and communicates with the FPTA digital interface through both the dedicated 32-bit interface on the SBC67 as well as the DIG-32 Omnibus module. The DSP system can be seen in figure 1. As shown in the figure, there are two paths from the DSP to the FPTA, the digital path along which the candidate configurations are downloaded and an analog path which is used when stimulating the FPTA and recording the response for evaluation on the DSP.

2.2 Software Architecture

The software implementing the genetic algorithm is designed around a protocol called DSP/BIOS designed by TI. DSP/BIOS is a realtime kernel which allows task

¹Innovative Integration is working hard on providing USB drivers for Windows 2000.

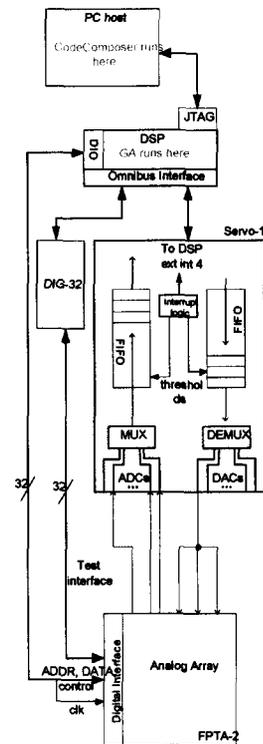


Figure 1: The DSP EHW platform showing DSP with associated I/O modules and the FPTA

scheduling and synchronization as well as configuration of the software architecture and advanced debugging features. The task scheduling ability is used to configure software as well as hardware interrupts. These interrupt routines can interact in a multi-tasking environment with the use of several levels of priority. Configuration of the processor and memory mapping for linking compiled code is as simple as specifying what memory segment to use for each 'section' of code. This configuration interface is used extensively.

In addition to configuration, DSP/BIOS allows debugging of the program running on the DSP by transferring register and memory contents to the host PC while the processor is temporarily halted.

The evolutionary program developed to run on this platform is called *ga1* and is broken into two main tasks, main and SupervisorTask. The main function is invoked by the task manager at DSP initialization. Some initialization is performed in main and when it exits the task manager schedules SupervisorTask to run. Super-

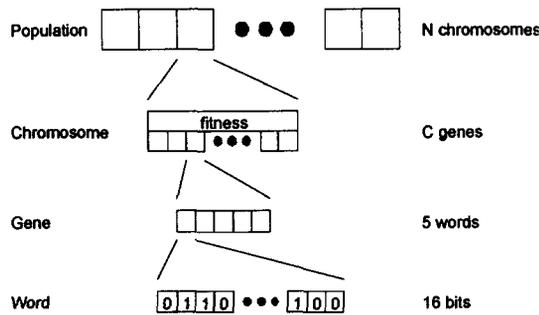


Figure 2: The structure of the population is shown as a collection of individuals. Each individual has a fitness and configuration data. The configuration data is broken into genes and further subdivided into five 16-bit words which correspond to the 80 configurable bits for each cell of the FPTA.

visorTask is the main routine for performing the GA.

3 Genetic Algorithm

As stated above, the GA basically iterates over a genome consisting of candidate configuration, making modifications between subsequent generations. The genome structure is shown graphically in figure 2 and is a population P of N chromosomes. Each chromosome contains a fitness value, and a set of genes corresponding to physical cell structures on the FPTA. Each gene consists of 80 configuration bits, which are split into five 16-bit words.

The genetic algorithm employed in version 1.0 of the evolutionary code is described in pseudo-code below.

Inputs:

t_s sample time

$R(t_s)$ the current response of the chip

$W(t_s)$ the desired response of the chip

$S(t_s)$ a stimulus to use

n the number of samples

P the population

N the population size

F_t the target fitness

C Number of genes to use (how much of the chip to utilize)

P_c Crossover probability

P_m Mutation probability

P_e Elite percentage to save

Outputs:

F_0 the fitness of best individual for each generation

$X(C)$ the final chromosome corresponding to the solution (chip configuration)

Initialize(P, N)

do{

 for ($i = 0; N$) {

 download $P(i)$

 stimulate the circuit $S(t_s)$

 evaluate the response $W(t_s)$

 }

$F_0 = \text{Sort}(P)$

 Select elite individuals(P)

 two_point_crossover(P)

 mutation(P)

}until ($F_0 \leq F_t$)

Sort: The sort function is implemented as the standard quicksort algorithm, sorting to low to high values of fitness. Since the population is implemented as an array of pointers, the memory overhead for small populations is not prohibitively expensive.

Crossover: The crossover function selects some percentage of the individuals and for those individuals selects two bit indices and swaps the sections between those two indices between two individuals. The selection of the two individuals to cross over is based on fitness. This is generally known as the *wheel* method because it normalizes the sum of all selection probabilities to 1 and then places each individual into a range of probability between 0 and 1, with the more fit individuals getting a

larger fraction of the available probability space. A random real number is then chosen between 0 and 1 and the corresponding individual is chosen. This allows that more fit individuals get selected for crossover more often.

The actual crossover of two individuals is done by considering the two individuals as a configuration bitstreams and by selecting a start and a stop index (two-point crossover) and by swapping the bits from the two individuals. No consideration is given to gene structure during this operation.

Mutation: The mutation function essentially loops a number of times corresponding to $P_m \times N$ and on each pass chooses an index into the configuration bitstream and flips it. This does not enforce that the same bit doesn't get flipped twice, but in doing so the function runs in $O(n)$ time rather than $O(n^2)$.

3.1 Programming

The FPTA uses a synchronous programming interface with control signals CLEAR, RESET, ST, WR, DATA[15:0] and ADDR[8:0] and a clock input CLK. The interface is synchronized on CLK which must be synchronized with the other control signals.

The programming function for chromosome i follows this procedure:

```
/* Reset the configuration logic */
Raise the CLEAR line
Lower the RESET line
Cycle CLK
/* Output ADDR/DATA pairs for
   programming */
for (n=0; n<C; n++){
  for(word=0; word<5; word++) {
    DATA = pop[i].gene[n].geneData[word]
    ADDR = n * word
    Raise ST and WR
    Assert DATA and ADDR lines
```

```
    Cycle CLK
    Lower the ST and WR lines
    Cycle the CLK
  }
}
```

In order to minimize the time taken to download a chromosome hand-coded assembly language has been written to access the memory-mapped I/O for the DIO which eliminates layers of system driver function calls.

3.2 Methodology of the stimulus/response cycle

Each individual in the population is evaluated in a stimulus/response cycle performed as part of the GA. The cycle consists of a waveform of some specified function $W(t_s)$ being input to the FPTA and the output of the chip $R(t_s)$ being sampled n times and recorded as the response to be analyzed under the fitness criteria. This cycle is performed with the use of the SERVO-16 module in an interrupt driven mode. The method used to stimulate the circuit is derived from the architecture of the SERVO-16 and its FIFO memory structures. There are two FIFO memories, one for input and one for output, as seen in figure 1, which are 32 bits wide and 256 entries deep. The 32 bits in each FIFO location are filled by two 16-bit samples, either input or output, an implication of this is that inputs and outputs are enabled in contiguous pairs (0:1, 2:3, etc.). The SERVO-16 control logic samples all enabled ADC pairs and updates all enabled DAC pairs at the same time, t_s , at some rate, R_s . The inputs/outputs are multiplexed to/from the respective FIFOs during the intervening sampling period. When an ADC pair is sampled, the lower numbered input is mapped to the low order 16-bits and the higher to the higher order 16 bits of the FIFO. If more than one pair are enabled the lower numbered pair is written to the FIFO first, followed by the next higher number, and so on. If the FIFO fills up then the next and subsequent samples are lost. When reading the ADC FIFO, if there are no entries in the FIFO, the last entry is read repeatedly. Similarly, if too many words are written to the DAC, the last samples are lost and if there are no entries in the FIFO, the last entry is output multiple times.

Interaction between the SERVO-16 and the DSP is initiated either when the DSP writes/reads data to/from the SERVO-16 or when the SERVO-16 issues an interrupt to the processor. This is because the DSP does not have access to the amount of data stored in the FIFOs and the only way to know the status of the FIFOs is to either reset the whole board or receive an interrupt from the SERVO-16. Interrupts are issued by the SERVO-16 based upon the number of entries in the one or both of the FIFOs based on one of three conditions, ADC_FIFO_HI, DAC_FIFO_LO or the logical OR of the two. The algorithm is currently written so that an interrupt occurs on DAC_FIFO_LO, which triggers when there are less than FIFO_THRESHOLD (currently 40) samples left in the DAC FIFO. When the interrupt occurs the interrupt service routine `analog_isr` is scheduled by the task manager and performs the FIFO updates. Since there are only 256 entries in the FIFO, a problem occurs when waveforms greater than 216 are employed. In this case there is a special section of code which writes and reads segments of the waveforms to the DAC and from the ADC

A time optimization is made by analyzing the $R(t_s)$ of one individual during the time the processor is waiting for the stimulus/response cycle for the next individual to complete. The cycle time is determined by the type of input, $S(t_s)$.

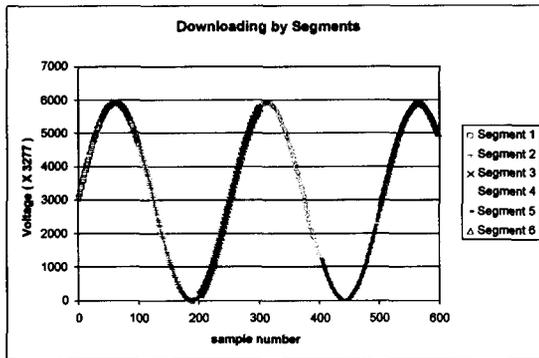


Figure 3: Showing Download segments, note that the minimum t_s is $10\mu s$.

Implications to the code: The `analog_isr` function is mapped to interrupt 4 in the configuration database file, `ga1.cdb`. The value of FIFO_THRESHOLD is set to 40.

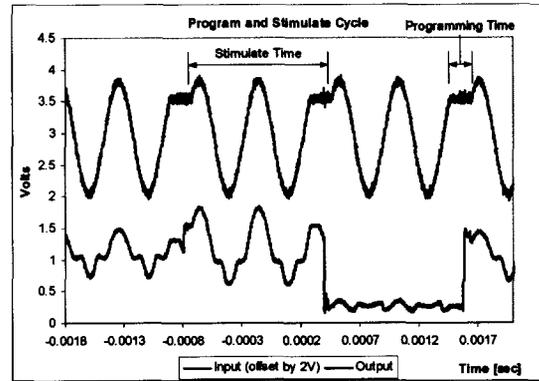


Figure 4:

4 Example Experiment

To show how the GA can be applied with this system can be demonstrated by a simple search for a configuration which implements a half-wave rectifier. In this case we choose the inputs to the algorithm as follows: $C = 2$, $N = 100$, $S(t) = \sin(t)$, $F_t = 4500$, $n = 50$, $t_s = 10\mu s$, $P_e = 0.2$, $P_m = 0.04$, $P_c = 0.7$, and

$$F = \sum_{t_s=0}^{n-1} \begin{cases} R(t_s) - S(t_s) & \text{for } (t_s < n/2) \\ R(t_s) - V_{\max}/2 & \text{otherwise} \end{cases}$$

where V_{\max} is the peak voltage of the input sine wave.

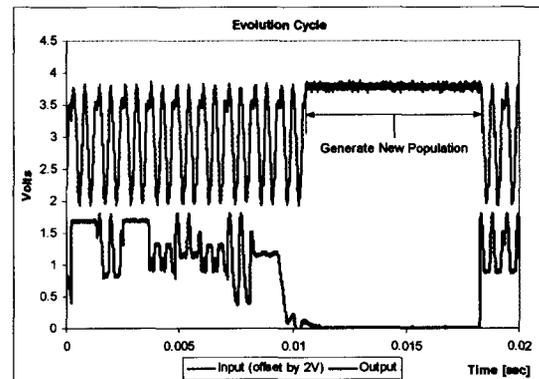


Figure 5:

5 Discussion

5.1 Representation of the configuration bitstream

The current representation of the configuration bitstream and subsequent implementations of the Crossover and Mutation functions does not consider that certain groups of bits only have a limited number of useful configurations. For instance, if we consider a group of 4 bits and enumerate the 16 different combinations we might discover that 11 of them appear to be indistinguishable. In that case we can reduce the search space by only allowing 6 different combinations of those 4 bits. The biological equivalent of the useful combinations is called an allele. Reducing the search space by including these alleles would most likely greatly improve the performance of the algorithm.

The performance of the algorithm is measured not only by the number of generations before convergence, but since one of the goals of this project is to provide fast evaluation, the evaluation time of the algorithm is also a metric of interest.

5.2 Code placement

The DSP/BIOS configuration tool is used to designate where the linker can locate code segments. There are 128KB of internal SRAM on the processor which can be split into two sections, Internal Program RAM (IPRAM) and Internal Data RAM (IDRAM). In addition, there are 16MB of external SDRAM, see figure 6. The default is to allow only program code to be stored in IPRAM, code or data to be stored in IDRAM and only data to be stored in SDRAM. However, when the code becomes too large to fit into the available internal RAM, we move sections to SDRAM. This impacts the speed of the algorithm because accesses to SDRAM take two clock cycles versus only one for internal RAM. In order to minimize the amount of internal memory used, a trade-off is made to place the main program (`.text`) into internal memory and placing large arrays such as `population[]` and `newPopulation[]` into the external SDRAM memory. The impact of these decisions directly affects the evaluation time of the algorithm, for instance, the FPTA programming cycle can take as little as $540\mu s$ when the pro-

gram is loaded in internal ram (IPRAM), but increases to $594\mu s$ if placed in external RAM (SDRAM).

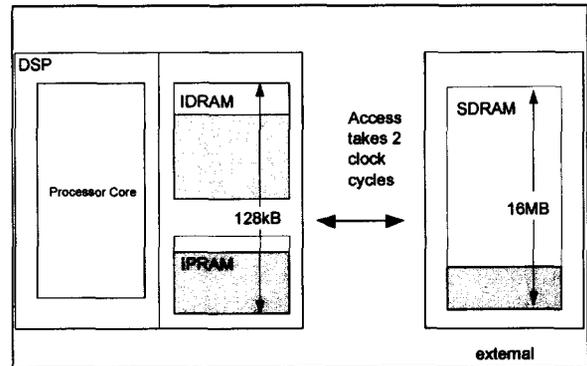


Figure 6: A graphical representation of the memory map. The 128kB internal RAM is divided into two segments, Program and Data memory, the external memory can be used for either.

6 Conclusion

We have shown that by using a stand-alone DSP and an FPTA we can gain by 3 to 4 orders of magnitude in evolution time (experiment time) over state-of-the-art evolution done using simulation. We have also demonstrated that code-optimizations such as hand-coded assembly instructions and careful data/code placement can have a significant effect on the performance of the algorithm.

Acknowledgment

The research described in this paper was performed at the Center for Integrated Space Microsystems, Jet Propulsion Laboratory, California Institute of Technology and was sponsored by the National Aeronautics and Space Administration and Defense Advanced Research Projects Agency (DARPA).

References

- [1] A. Stoica, R. Zebulum, D. Keymeulen, R. Tawel, T. Daud, and A. Thakoor, "Reconfigurable VLSI Architectures for Evolvable Hardware: from Experimental Field Programmable Transistor Arrays to Evolution-Oriented Chips", IEEE Transactions on

VLSI Systems, Special Issue on Reconfigurable and Adaptive VLSI Systems, February 2001.

[2] Innovative Integration model SBC6x stand-alone DSP board, 805.520.3300, www.innovative-dsp.com

[3] OMNIBUS User's Manual Rev 1.12, p144.