Published in IET Computers & Digital Techniques
Received on 6th July 2006
Revised on 21st September 2007
doi: 10.1049/iet-cdt:20070027

ISSN 1751-8601

Hardware architecture for high-speed real-time dynamic programming applications

B. Matthews I. Elhanany
Electrical and Computer Engineering Department, University of Tennessee, Knoxville, TN, USA
E-mail: itamar@ieee.org

Abstract: A novel hardware architecture for performing the core computations required by dynamic programming (DP) techniques is introduced. The latter pertain to a vast range of applications that necessitate an optimal sequence of decisions to be obtained. An underlying assumption is that a complete model of the environment is provided, whereby the dynamics are governed by a Markov decision process. Existing DP implementations have traditionally focused on software-based mechanisms. Here, the authors present a method for exploiting the inherent parallelism associated with computing both the value function and optimal policy. This allows for the optimal policy to be obtained several orders of magnitude faster than traditional software implementations, establishing the viability of the approach for demanding, real-time applications. The well-known rental car management problem has been studied as a benchmark for which a field-programmable gate array-based implementation was designed. The results highlight the advantages of the proposed approach with respect to the execution speed and the scalability properties.

1 Introduction

A wide range of engineering applications, ranging from communications [1–3], robotics [4, 5], to automotive [6] systems, employ Markov decision processes (MDPs) as an underlying formalism to determine the optimal policy, or control scheme, in a dynamic stochastic environment. An MDP is defined as a quadruple $\langle S, A, P_A, R\rangle$, where S denotes a finite state space, A contains all possible actions that can be taken at a particular state, P_A represents the probability transition function $|S| \times |S| \times A \rightarrow [0, 1]$ and R represents the mapping of state-action pairs to rewards, $R : |S| \times A \rightarrow \mathbb{R}$. The MDP quadruple serves as a perfect model for an application space, whereas the actual planning, which involves determining an optimal set of actions that must be taken to accumulate maximum reward, is referred to as a dynamic programming (DP) problem. Existing research efforts [7, 8] have traditionally focused on software-based realisations of DP techniques to determine the set of mapping of states to actions, or policy, required to derive the optimal control utilising traditional general instruction processors. In contrast, real-time systems often require custom solutions in order to meet stringent timing requirements often associated with these applications. To apply DP techniques in the framework of real-time applications, we propose an architecture for solving DP problems in custom digital hardware, exploiting parallelism not possible in software-based realisations. This facilitates the implementation of large-scale real-time decision-making systems that operate under uncertainty.

Recent advancements in VLSI technology have lead to an increase in logic density and on-chip memory resources availability. An $O(|S|^2\ A)$ [9] memory requirement, principally because of the probability transition function, restricts DP algorithms that possess a large state set from being used in many applications. This restriction is, in principle, because of the limited amount of on-chip storage available in current devices. Exploiting parallelism that is an inherent attribute of DP algorithms, necessitates large, parallel memory in order to store the probability transition function. Fortunately, existing field-programmable gate array (FPGA) devices [10, 11] now provide multiple megabits of distributed on-chip

SRAM, as well as an abundance of embedded fast multipliers that can be used to exploit the parallelism involved in DP computations. Although the memory requirement is not eliminated by the architecture proposed in this paper, we do present techniques to reduce the memory bandwidth burden while, at the same time, providing a computational speedup that is several orders of magnitude faster than traditional software-based schemes.

The rest of the paper is organised as follows. In Section 2, we review the fundamentals of DP and outline the policy iteration algorithm and its hardware realisation. In Section 3, value function optimisation method is presented, which is crucial to in reducing the memory requirements. A novel hardware architecture for the realisation of DP techniques, is presented in Section 4. FPGA-based implementation results, accentuating the performance and scalability attributes of the proposed approach, are presented and discussed in Section 5. Finally, in Section 6, the conclusions are drawn.

2 Overview of DP

DP is considered to be a collection of algorithms that, given a perfect model of the environment as represented by an MDP, allows for an optimal policy, or mapping of states to actions, to be determined. In order to establish an optimal policy, the probability of transitioning to each possible next state, s', given any state and action pair, s and a, is given by

$$P^a_{ss'} = \Pr[s_{t+1} = s' | s_t = s,\ a_t = a] \tag{1}$$

To complement the above, we define a model such that the reward obtained one step after transitioning from state s to state s', by taking action a, to be r_{t+1}, such that its expectation can be represented in the form

$$R^a_{ss'} = E[r_{t+1} | a_t = a,\ \ s_t = s,\ \ s_{t+1} = s'] \tag{2}$$

The reward function provides information that can be used to determine the value of the next state, given that action a is taken. The long-term value associated with a given state is provided by the state-value function. The latter is defined as the expected sum of discounted rewards, where γ is the discounting factor, given that the system begins in state s and follows policy π, such that [12]

$$V^{\pi}(s) = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right] \tag{3}$$

A common goal in DP application is to derive an optimal policy that maximises the value function

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{4}$$

Utilising (2) and (4), the optimal value function, expressed in terms of the transition probabilities and reward function, is given by

$$\begin{aligned} V^*(s) &= \max_a E_{\pi^*}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s,\ \ a_t = a\right] \\ &= \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')] \end{aligned} \tag{5}$$

With the optimal value function defined, we next outline a common methodology, known as generalised policy iteration (GPI), for arriving at the optimal policy. Policy iteration involves an evaluation phase, where a policy, $\pi(s)$, is evaluated to determine an optimal value function, and an improvement phase, that is employed to make the current policy, $\pi(s)$, greedy with respect to the value function. During the evaluation phase, the value function improves according to the current policy. While in the improvement phase the policy is updated according to the current value function. As both the policy and value function improve over successive iterations, they are guaranteed to converge to the optimal value function and the optimal policy [13]. This approach is illustrated in the algorithm in Fig. 1 [14]

```
Policy Iteration
1. Initialization: Arbitrarily initialize $V(s)$ and randomly select a policy, $\pi(s)$.
2. Policy Evaluation
    while OptimalValueFunction is not true
        for each $s \in |S|$:
            $V_{prev}(s) = V(s)$
            $V(s) = \sum_{s'} P^{\pi(s)}_{ss'} \left[R^{\pi(s)}_{ss'} + \gamma V(s')\right]$
            if $|V_{prev}(s) - V(s)| < \varepsilon$
                OptimalValueFunction is true
            else
                OptimalValueFunction is false
3. Policy Improvement
    OptimalPolicy is true
    for each $s \in |S|$:
        $\pi_{prev}(s) = \pi(s)$
        $\pi(s) = \arg\max_a \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V(s')]$
        if $\pi_{prev}(s) \neq \pi(s)$
            OptimalPolicy is false
    if OptimalPolicy is false,
        repeat Policy Evaluation
```

Figure 1 *Policy iteration algorithm*

Table 1 Generic memory requirements

	States	Action	Bits/value	Memory required
transition probability	225	11	12	6.683 Mbits
reward function	225	11	9	5.012 Mbits

3 Optimisation for hardware design

Prior to discussing the hardware realisation of the policy iteration algorithm, we first outline the associated limitations in terms of storage requirements. It is apparent that both the transition probabilities, $P^a_{ss'}$, and expected reward function, $R^a_{ss'}$, mandate an $O(|S|^2\ A)$ memory requirement. To illustrate the storage impact of this requirement, we first consider an application comprising of 225 states, and restrict the number of actions that can be taken at each state to be 11. Furthermore, the transition probabilities and rewards are represented using 12-bit and 9-bit values, respectively. The aggregate memory requirement, as shown in Table 1, needed to store the transition probabilities and reward function values is 11.7 Mbits, which exceeds the on-chip memory available in current FPGA devices.

In order to efficiently map the policy iteration algorithm to hardware, it is imperative that the storage requirements be reduced. To facilitate such reduction, we first rewrite the policy improvement expressions as

$$\begin{aligned}\pi(s) &= \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')] \\ &= \arg\max_a \left[\sum_{s'} P^a_{ss'}R^a_{ss'} + \sum_{s'} P^a_{ss'}\gamma V(s')\right] \\ &= \arg\max_a \left[G(s, a) + \sum_{s'} H^a_{ss'}V(s')\right] \end{aligned} \tag{6}$$

where the value of the state-action pair, $G(s, a)$, is given by

$$G(s, a) = \sum_{s'} P^a_{ss'}R^a_{ss'} \tag{7}$$

for which the set of discounted transition probabilities, $H^a_{ss'}$, is given by

$$H^a_{ss'} = \gamma P^a_{ss'} \tag{8}$$

We apply a similar technique in an effort to reduce the storage requirements characterising the improvement phase. To that end, the state-value function may be expressed in the form

$$\begin{aligned} V(s) &= \sum_{s'} P^{\pi(s)}_{ss'}\left[R^{\pi(s)}_{ss'} + \gamma V(s')\right] \\ &= G(s, \pi(s)) + \sum_{s'} H^{\pi(s)}_{ss'}V(s') \end{aligned} \tag{9}$$

The values for $P^a_{ss'}$ and $R^a_{ss'}$ are provided by the MDP such that the computation of $G(s, a)$ can be performed offline in order to reduce the associated storage requirements from $O(|S|^2A)$ to $O(|S|A)$

Unfortunately, it is infeasible to similarly reduce the number of unique transitional probabilities from $O(|S|^2A)$. However, we can decrease the number of bits required to store each unique probability value. We selected a 7-bit representation as sufficient resolution for these values. In representing the probability as a discrete set of potential values, some error is introduced. This error is less than one-percent per computation and is distributed evenly over what is already an inherent approximation. Moreover, having made the storage requirements for the probabilistic values more moderate, we can now offer a greater dynamic range for actual reward values. Exploiting the aforementioned observations, the total memory requirement, as presented in Table 2, is reduced from 11.7 to 3.89 MBits, which is suitable for FPGA-based implementations.

The final observation to be made does not pertain to the storage properties, but rather to the reduction in computation requirements resulting when utilising the offline processing method to condense the reward

Table 2 Reduced memory requirements

	States	Action	Bits/value	Total memory
transition probability	225	11	7	3.89 Mbits
reward function	225	11	18	44.55 kbits

function. Recognising that the product of the discount parameter, γ, and transitional probabilities, $P^{a}_{ss'}$, can be computed offline, we obtain a smaller number of on-chip multipliers required. This further decreases power, area and latency associated with the hardware realisation of the DP policy iteration algorithm.

4 System architecture

The hardware realisation of the policy iteration algorithm, as shown in Fig. 2, offers a modular architecture with respect to the number of states that can be addressed in parallel. Its goal is to produce an optimal action selection in real-time. The architecture is comprised of a parallel value estimation module, an policy evaluation primitive and a control unit that alternates between policy evaluation and policy improvement modes to support the generalised policy iteration framework. The scalability property of this architecture is derived from the ability to add value-estimation primitives, with a fixed latency cost, to the parallel value estimation module.

In establishing the foundations of DP, the generalised policy iteration algorithm outlined an evaluation phase that optimises the value function, and an improvement phase that updates the control policy. For both phases, iteration across the state space is differentiated primarily by the number of actions required to perform an update. For the improvement phase, the policy update requires the value function to be computed for all possible actions, with the outcome resulting in storage of the action that produces the maximum value. The evaluation phase only requires the value function to be computed for a single action, dictated by the policy, with the update resulting in the storage of the computed value.

For efficiency, we implement a single, parallel value estimation module that receives the state-action pair to be evaluated from the DP control module. During the policy improvement phase, the state value supplied by the DP control module is fixed for every potential policy action available, whereas for the evaluation phase the state value changes concurrently with action value dictated by the selected control policy, $\pi(s)$. From the perspective of the value estimation module, there is no distinction between evaluation and improvement. Such distinction is only made at the policy evaluation primitive. Although the evaluation and improvement phases could be constructed using seperate modules, this would not be an efficient solution as the hardware resource requirement would double, given the addition of a second parallel value estimation module.

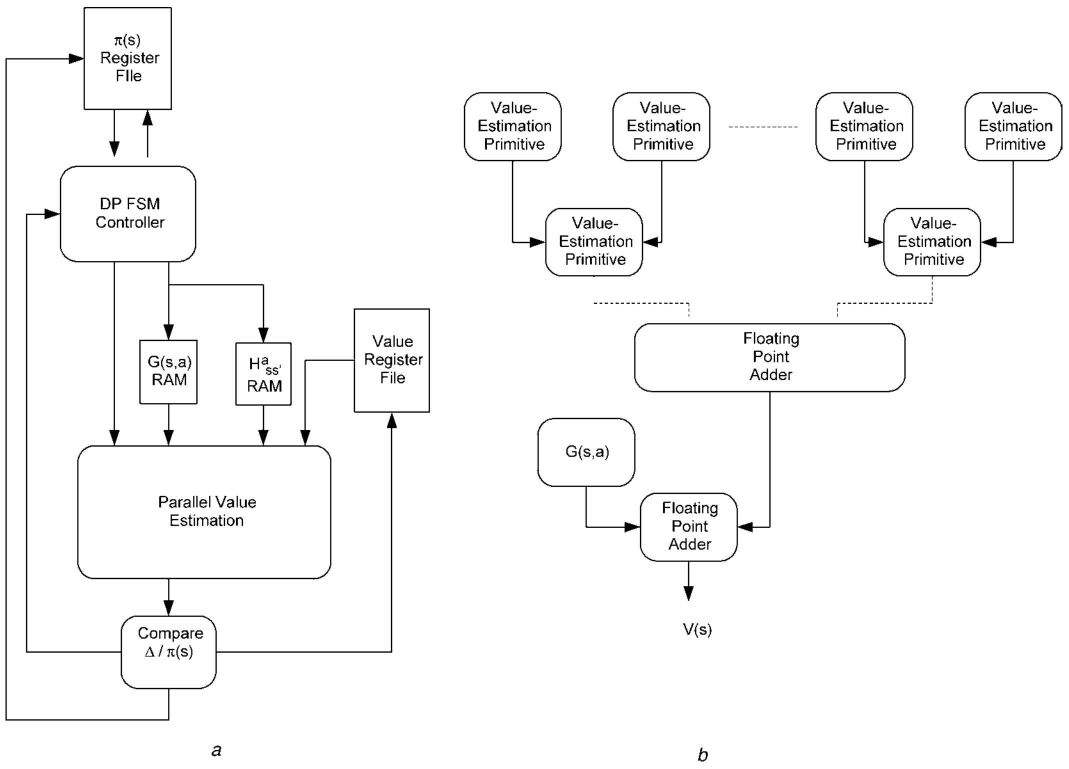

Figure 2 *Hardware realisation of the policy iteration algorithm*

a Policy iteration architecture for FPGA implementation
b Parallel value computation module

The parallel value computation, presented in Fig. 2, is constructed using S concurrent value-estimation primitives, depicted in Fig. 3, to compute the product of the discounted transition probability, $H^a_{ss'}$, and the potential next state value, $V(s')$, using 18-bit floating point multipliers. The result of the S parallel multiplications is forwarded to a tree of floating point adders, with a depth of log $(|S| + 1)$, for summation. In targeting a broad range of applications, the architecture must be structured in a manner than provides scalability in terms of the number of states that can be supported. To do so, the selection of floating point multipliers and adders, as opposed to fixed-point units, was made in an effort to maintain fixed delay and constant structure while retaining a high degree of accuracy.

The next state values, $V(s')$, are stored in S 18-bit registers that are all accessed in parallel. In an effort to reduce memory requirements, the $H^a_{ss'}$ values are 7-bit encoded and are distributed at equidistant points in the range of [0,1]. These values are converted to 18-bit floating point values, by the $H^a_{ss'}$ decode unit, prior to being forwarded to the floating point multiplier. It should be noted that encoded values must be stored in S RAM units for parallel access by the S value-estimation primitives. Each of the of S memory units corresponds to a single next state value, S', with its address represented by the current state-action pair.

In connection with recent work that has examined the effects of delayed information within the MDP framework [15], it is appropriate to acknowledge that our architecture is subject to similar concerns given the pipelined nature of the parallel value estimation module, $V(s)$. This pipelining results in an update delay that directly corresponds to the associated latency of the value estimation module and the policy evaluation primitive. In our architecture, value estimates are not available instantaneously, which can result in convergence delay. For this reason, the system will select an action based on previous values, rather than the current ones. Given the significant speedup provided by a hardware implementation, the reduced convergence rate, as will be demonstrated, is minor.

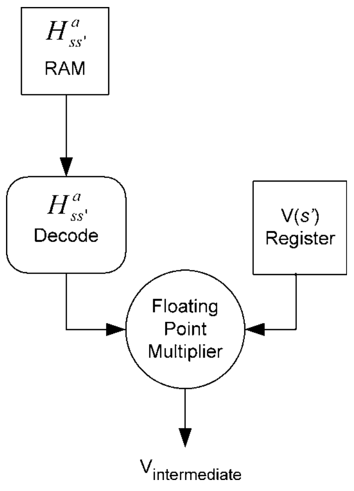

Figure 3 *Computation unit structure*

The result of the parallel value computation is passed to an evaluation module, which, depending on whether it is operating in the evaluation or improvement phase, is responsible for updating either the value function or the control policy, respectively. In addition to updating the value function and control policy, it must also determine whether they are close enough to the optimal solution. Addressing the evaluation mode first, it is required that at the beginning of each iteration a flag bit is set so as to indicate that the optimal value function has been found. As we iterate across each state, the difference between the previously registered value and the newly computed one is obtained. The result is then compared to a small value, ε, using a floating-point comparator module, to determine whether an optimal value has been reached. If any state which produces a value that is not optimal the flag bit, indicating an optimal value function, is cleared. Upon completion of the calculations pertaining to the final state, the DP controller tests the flag bit. If the latter is set, the controller switches to a policy improvement mode, otherwise the evaluation module repeats its iterative policy evaluation process.

The switch from policy evaluation to policy improvement represents a significant change in functionality of the evaluation module. In policy improvement mode, the adder performs no addition operations and merely serves as a pipeline stage, with its output forwarded to a maximum value module. The latter comprises of a register and a comparator. The contents of the maximal value register are updated once a newly computed result is found to be greater than the existing one. Upon sweeping through the action set for a given state, the action that yielded the maximal value is stored in the policy RAM.

If, upon receiving the final sample of a prior evaluation, it is determined that an optimal value function has been reached, the pipeline is flushed and the system transitions to an improvement state during which the process again iterates over all possible states. However, state transitions do not occur until the results for all possible actions that can be taken from the current state, have been evaluated. Once the final state-action pair has been identified, the system flushes its pipeline and transitions to an evaluation state, the goal of which is to determine whether an optimal policy has been found. If the evaluation process determines the policy to be optimal, a flag is asserted, the system flushes the pipeline and waits for future instructions.

5 Implementation results

In this section, we employ the DP architecture presented in the prior section to solve the rental car management problem introduced in [16]. In this problem, a manager is responsible for distributing cars between two rental car locations, each containing a maximum of 11 cars. For each car rented at either location, the manager is credited \$10 by the national company. Vehicles arrive and depart the car rental locations according to a geometric distribution, Pr $\{n\ arrivals\} = (1-\lambda)\lambda^{n-1}$, where n is a random variable denoting the number of arriving, or departing, vehicles. The arrival probabilities, λ_{arrival}, for locations 1 and 2, are 0.3 and 0.5, respectively. Correspondingly, the departure probabilities, $\lambda_{\text{departure}}$, for locations 1 and 2, are 0.3 and 0.25. It should be noted that, since each location is limited to 11 cars, any arriving cars that exceed the site limit are discarded. The manager is permitted to move cars between the two locations at a cost of \$2 per car, with a limit of 5 cars that can be moved in a single night. DP formalism is applied to determine the optimal policy in an effort to maximise the manager's profit across the two locations.

To solve the aforementioned problem, the DP architecture has been coded in Verilog HDL with the implementation targeting an Altera Stratix II EP2S180 FPGA device [10]. The design is comprised of 121 states with a total of 11 potential actions that can be taken from each state. The complete system implementation achieved a post-fit operating frequency of 103 MHz (9.7 ns). The design required 47 453 adaptive logic modules (ALMs), or 66% of the ALMs available on the target FPGA device. Since the architecture requires a single RAM such that values for each of the 121 states can be accessed in parallel, the design consumed 493 (64%) of the M4K RAMs available. Additionally, a single M512 RAM unit was employed for storing $G(s, a)$ values. The total memory requirement was 1.772 Mbits, or 18% of that available.

In order to establish the coarse performance gain that can be achieved when implementing the hardware design, a software-based solution to the rental car management problem was written. When simulated, the policy iteration algorithm converged after an average of 33 evaluation iterations and 4 improvement iterations with an execution time of 9.174 s. The optimal policy and value function are shown in Figs. 4 and 5, respectively.

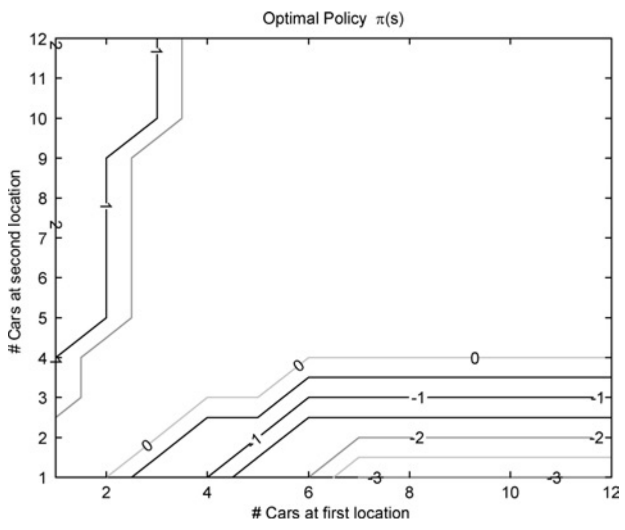

Figure 4 *Optimal policy for rental car management problem*

Given that the hardware design computes values in parallel, the equivalent hardware run time can be determined by first establishing the system latency. The latency contributed by the floating-point multiplication, l_{mult}, is five clock cycles. The latency due to the floating point adder, l_{add}, and the floating-point comparator units are four and one clock cycles, respectively. The final latency component to be considered is a result of miscellaneous pipeline registers that are needed to ensure maximum throughput, l_{pipe}, and is comprised of four clock cycles.

The hardware runtime can now be established in terms of the system latency, policy improvement iterations (R_{i}), policy evaluation iterations (R_{e}), the number of states ($|S|$), actions (A) and clock periods (t_{clk}). The number of clock cycles comprising a single

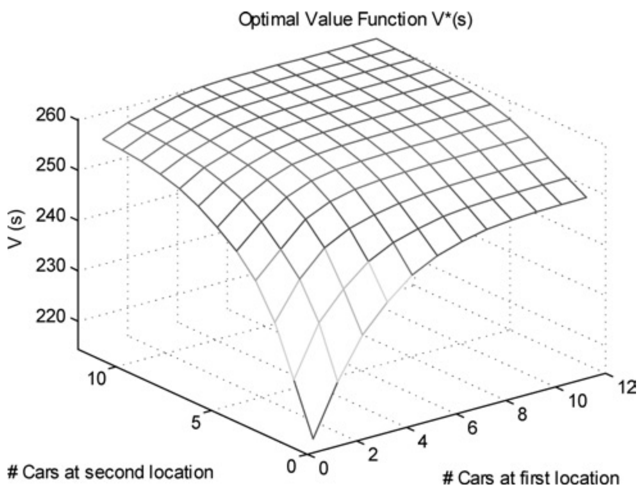

Figure 5 *Optimal value function for rental car management problem*

policy evaluation phase is

$$c_{\text{eval}} = |S| + l_{\text{mult}} + (\log_2(|S|+1) \times l_{\text{add}}) + l_{\text{add}} + l_{\text{compare}} + l_{\text{pipe}} \quad (10)$$

In the improvement phase, the clock cycle requirement must include the value computation latency for every action, which is contrary to the evaluation phase for which only one action is selected. As such, the time to complete a single improvement phase is given by

$$c_{\text{improve}} = (|S| \times A) + l_{\text{mult}} + (\log_2(|S|+1) \times l_{\text{add}}) + l_{\text{add}} + l_{\text{compare}} + l_{\text{pipe}} \quad (11)$$

The optimal policy and value function hardware computations can be expressed as a function of the evaluation phase and improvement phase cycle requirements, c_{eval} and c_{improve}, such that

$$t_{\text{total}} = R_{\text{i}} \times ((R_{\text{e}} \times c_{\text{eval}}) + c_{\text{improve}}) \times t_{\text{clk}} \quad (12)$$

Further, we can employ (10) and (11), establishing the hardware evaluation cycles, c_{eval}, to be 163 clock cycles and the number of improvement iteration cycles, c_{improve}, to be 1373 cycles. From these, the aggregate time can be obtained by applying (12), such that the hardware computation time to solve the rental car management problem is 270.1 μs. Thus, the hardware-based solution yields four orders of magnitude improvement in timing when compared to the software-based realisation.

This considerable gain can be attributed to the considerable parallelisation associated in calculating the value function, which is in contradiction to the sequential processing inherent to the software-based approach. Notably, the policy-evaluation processing requirement has been reduced from an $O(|S|^2)$ processing requirement, for the software implementation, to an $O(|S| \times \log(|S|))$ processing requirement in the corresponding hardware implementation. Furthermore, the policy improvement phase results in a similar processing reduction, where the software processing requirement is now $O(|S|^2 \times A)$, compared to $O(|S| \times (|S|) \times A)$ for the hardware. Additional gains can be attributed to direct memory access, lack of instruction overhead (fetch, store, pipeline delay and so on), and exclusive access to the computation units. In reducing the computation time to the microseconds range, DP can now considered for an increased number of real-time applications.

6 Conclusions

In this paper, a novel hardware-based architecture was introduced for real-time DP applications. It has been demonstrated that the proposed architecture can yield a speed gain of several orders of magnitude when compared to software-based systems. To that end, the framework is suitable for a broad range of applications necessitating fast, real-time sequential decision making. FPGA implementation results were presented and discussed to emphasise the viability and scalability of the proposed architecture.

7 Acknowledgment

This work has been partially supported by the United States Department of Energy (DOE) research contract DE-FG02-04ER25607 and by Altera, Inc. research grant.

8 References

[1] JAVIDI T., TENEKETZIS D.: 'Sensitivity analysis for an optimal routing policy in an *ad hoc* wireless network', *IEEE Trans. Autom. Control*, 2004, **49**, (8), pp. 1303–1316

[2] HAAS Z., HALPERN J.Y., LI L., *ET AL.*: 'A decision-theoretic approach to resource allocation in wireless multimedia networks'. Proc. 4th int. workshop Discrete algorithms and methods for mobile computing and communications DIALM '00, 2000, pp. 86–95

[3] USAHA W., BARRIA J.: 'Markov decision theory framework for resource allocation in leo satellite considerations', *IEE Proc. Commun.*, 2002, **149**, (56), pp. 270–276

[4] LAROCHE P., CHARPILLET F., SCHOTT R.: 'Mobile robotics planning using abstract Markov decision processes'. Proc. 11th IEEE Int. Conf. Tools with Artificial Intelligence ICTAI '99, Washington DC, USA (IEEE Computer Society, 1999), p. 299

[5] FERGUSON D., STENTZ A.: 'Focussed processing of mdps for path planning'. 16th IEEE Int. Conf. Tools with Artificial Intelligence, ICTAI 2004, pp. 310–317

[6] KANG J., KOLMANOVSKY I., GRIZZLE J.: 'Approximate dynamic programming solutions for lean burn engine aftertreatment'. 38th IEEE Conf. Decision and Control, 1999, vol. 2, pp. 1703–1708

[7] PADBERG F.: 'On the potential of process simulation in software project schedule optimization'. 29th Annual Int., Computer Software and Applications Conf., COMPSAC 2005, vol. 2, pp. 127–130

[8] KIM S., LEWIS M., WHITE C.C.: 'Optimal vehicle routing with real-time traffic information', *IEEE Trans. Intell. Transp. Syst.*, 2005, **6**, (2), pp. 178–188

[9] KOTSALIS G., DAHLEH M.: ‘Model reduction of irreducible markov chains’. Proc. 42nd IEEE Conf. Decision and Control, 2003, vol. 6, pp. 5727–5728

[10] ‘Altera Stratix II technical documentation’, available at: http://www.altera.com

[11] ‘Xilinx Virtex-4 technical documentation’, available at: http://www.xilinx.com

[12] SUTTON R.S.: ‘On the significance of markov decision processes’. ICANN, 1997, pp. 273–282

[13] BERTSEKAS D.P., TSITSIKLIS J.N.: ‘Neuro-dynamic programming’ (Athena Scientific, 1996)

[14] BELLMAN R.E.: ‘Dynamic programming’ (Princeton University Press, 1957)

[15] KATSIKOPOULOS K., ENGELBRECHT S.: ‘Markov decision processes with delays and asynchronous cost collection’, *IEEE Trans. Autom. Control*, 2003, **48**, (4), pp. 568–574

[16] SUTTON R.S., BARTO A.G.: ‘Reinforecement learning: an introduction’ (Camrbridge, MA, MIT Press, 1998)