# A Symmetric Multiprocessor Architecture for Multi-Agent Temporal Difference Learning

Scott Fields, *Student Member, IEEE*, Itamar Elhanany, *Senior Member, IEEE*
Department of Electrical & Computer Engineering
The University of Tennessee
Knoxville, TN 37922
{sfields1, itamar}@utk.edu

***Abstract*— Temporal difference learning methods have been successfully applied to a wide range of stochastic learning and control problems. In addition to correctness, one metric of a technique's performance is its learning rate – the number of iterations required to converge to an optimal solution. The learning rate can be increased by using multiple agents that can share experience. In a software environment, the potential speedup from additional agents is limited, since adding agents significantly increases the burden of computation and/or hinders real-time processing. To address this problem, this paper presents a parameterized hardware model of a multi-agent system based on a shared-memory Symmetric Multiprocessor (SMP). To the author's knowledge, this is the first application of an SMP architecture to a multi-agent reinforcement learning system. The control model employed is a multi-agent variation of the Sarsa($\lambda$) algorithm. Several hardware optimizations schemes are investigated with respect to feasibility and expected performance. The system is modeled using a cycle-accurate simulation in SystemC. The results indicate that real-time learning rates can be significantly improved by employing the proposed parallel hardware implementation.**

## I. Introduction

Temporal difference learning methods [1] are among the most popular reinforcement learning (RL) techniques, primarily due to their simple implementation, low calculation requirements and favorable practical results. Through an iterative process, the algorithms solve for optimal action policies and value functions, allowing them to maximize the return over time. Many stochastic modeling and control problems can be solved using a reinforcement learning framework such that, given sufficient time and processing resources, an optimal solution can be found.

A key performance metric is the method's learning rate, as measured by either the number of trials/iterations or the elapsed time. One approach to increasing the learning rate is to employ multiple agents, allowing them to share perception, experience or policies [2]. For instance, one agent might interact with a real-world environment, while the others might interact with a simulator. Given that the simulations are representative of the environment, this has been shown to decrease the overall number of trials required to obtain the optimal solution. However, adding agents increases the computational complexity and can impact the learning time. The additional complexity and runtime can be a limiting factor when dealing with large state and/or action sets as well as real-time requirements.

A parallel implementation has the potential to overcome the difficulties of practical multi-agent systems. One approach in [3] involves using a parallel computer and message passing interface as well as a rigid partitioning of the problem domain. This work presents a different approach, using a shared-memory symmetric multi-processor (SMP) hardware architecture. The motivation for the use of an SMP architecture is to facilitate practical low cost, high performance parallel multi-agent implementations. Since a single agent's temporal difference calculations are relatively straightforward, an agent can be implemented in a single custom processing element (PE). The SMP architecture allows the agents to collaborate by providing them equal access to a shared memory space, and since each PE requires modest logic, an entire SMP system could be realized on a modern high-density FPGA. Moreover, strict partitioning of the problem domain is not required since the SMP architecture arbitrates between the agents.

This paper investigates the potential tradeoffs associated with an SMP-based multi-agent temporal difference learning system. The rest of the paper is structured as follows. In Section II the single-agent Sarsa($\lambda$) temporal difference learning algorithm is briefly described along with its multi-agent formulation. In Section III the benefits of multi-agent Sarsa($\lambda$) in a software environment are examined. Section IV discusses the concerns and possible optimizations pertaining to the implementation of the multi-agent Sarsa($\lambda$) in hardware. Section V presents a performance evaluation study on the hardware system model, and in Section VI conclusions are drawn and a discussion of future work is provided.

## II. The Single-Agent Sarsa($\lambda$) Algorithm

In the RL framework, the environment is modeled by a Markov decision process (MDP) in which the agent resides at a particular state $s_i$. The agent chooses an action $a$ from the available actions, and as a result the it moves to a subsequent state $s_j$ and receives a reward $r$. In episodic tasks, the agent continues selecting new actions until it arrives at a goal state. In choosing an action, the agent attempts to maximize its total reward, and in subsequent episodes the agent retains some knowledge of its prior actions. Since the state transitions can

Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s, $a$
Repeat (for each episode):
  Take action $a$, observe $r$, $s'$
  Choose $a'$ from $s'$ using policy derived from $Q$
    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    For all $s$, $a$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $s \leftarrow s'$; $a \leftarrow a'$
until $s$ is terminal

Fig. 1. Single-Agent Sarsa($\lambda$) Algorithm

Initialize $Q(s,a)$ arbitrarily, for all s, $a$
Initialize $e^i(s,a) = 0$, for all $i$, s, $a$
  Repeat (for each episode of each agent):
  Take action $a^i$, observe $r^i$, $s^{i\prime}$
  Choose $a^{i\prime}$ from $s^{i\prime}$ using policy derived from $Q$ (e.g., ε-greedy)
  $\delta^i \leftarrow r^i + \gamma Q(s^{i\prime},a^{i\prime}) - Q(s^i,a^i)$
  $e(s^i,a^i) \leftarrow e(s^i,a^i) + 1$
  For all $s$, $a$:
    $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
    $e^i(s,a) \leftarrow \gamma\lambda e^i(s,a)$
    $s^i \leftarrow s^{i\prime}$; $a^i \leftarrow a^{i\prime}$
until $s^i$ is terminal

Fig. 2. Multi-Agent Sarsa($\lambda$) Algorithm

be stochastic, the RL agent must balance exploitation and exploration in order to develop a reward-maximizing policy.

The single-agent Sarsa($\lambda$) on-policy control algorithm, detaile d in [1], maintains at its core a table of state-action values $Q(s,a)$, and a table of eligibility traces, $e(s,a)$. As a soft-max policy follows, the state-action values are updated by a step factor, $\alpha$, to more closely match the received rewards, $r$. An exhaustive interation over the state-action space is done in order to pass the update values backwards to previously-visited states based on a temporal factor, $\lambda$, and a discount factor, $\gamma$. The algorithm is outlined in figure 1.

Extending this single-agent construct into multi-agent Sarsa($\lambda$) can yield a number of different formulations, as detailed in [2]. In this particular implementation, each agent maintains its own states, actions, updates and eligibility traces. For the $i^{th}$ agent, these are $s_i$, $a_i$, $\delta_i$, and $e_i(s,a)$, respectively. This formulation uses identical policies for the multiple agents, though the policies are allowed to be independent (e.g. one could be greedy while the others $\varepsilon$-greedy). However, all agents update the *same* commonly-shared state-action table, $Q(s,a)$ [4]. The multi-agent algorithm is shown in figure 2.

In order to evaluate the different techniques, we refer to a unified task that is to be performed by each system. The task involves finding the shortest path through a $15 \times 15$ grid world. Each grid is mapped to one state, while the agent must travel from its starting state, positioned in the upper left corner, to a goal state at the bottom right corner, as depicted in figure 3.

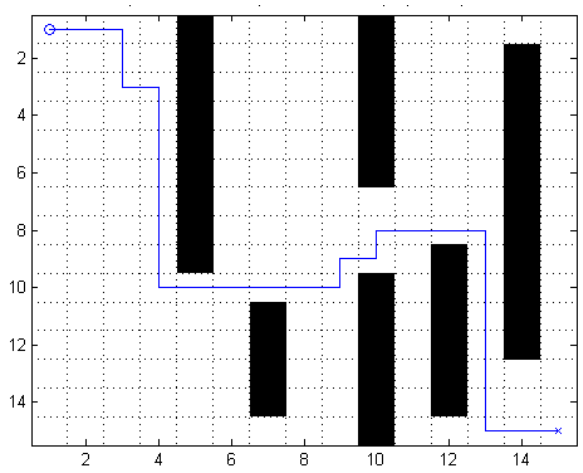


Fig. 3. The grid world is shown with obstacles and an optimal path to the goal.

The agent can take any of four possible actions corresponding to the four cardinal directions. A number of obstacles have been placed in the grid world. If the agent attempts to position itself off the grid or, alternatively, attempts to move into an obstacle, no change in its position occurs. A positive reward is given upon reaching the goal state, which also indicates the end of an episode. Trial and error was used to parameterize a single-agent for this task, and the same parameters were used for the multi-agent formulation: $\varepsilon$ = 0.85, $\gamma$ = 0.75, $\alpha$ = 0.5, and $\lambda$ = 0.5. This grid world, along with one of its optimal solutions (comprising of 32 steps), is illustrated in figure 3.

## III. SOFTWARE PERFORMANCE OF MULTI-AGENT SARSA($\lambda$)

Implementing the Sarsa($\lambda$) algorithm in software is relatively straightforward. The increase in learning rate due to the deployment of multiple agents can be evaluated by plotting the cumulative steps of the first agent versus the episode, as shown in figure 4. The derivative of the curves shown reflect the improvement rate for each episode. For a single agent, fewer steps are required to converge to the optimal solution. Thus, the solution can be obtained in fewer episodes.

As the number of agents increases, the system must process a greater number of steps, and the processing time increases respectively. As depicted in figure 5, these increases are sub-linear, since the agents are able to draw on each other's experience thereby reducing the uncertainty in the implications of their actions. In other words, there is overlap in the experiences of the various agents, so not all of the additional experience contributed by an additional agent is useful. The runtimes here reflect a 32-bit integer implementation of the algorithm that limits the eligibility traces to those visited during the past 12 steps. With the $15 \times 15$ grid world problem, the entire grid fits into cache and the integer math and memory accesses are relatively fast.

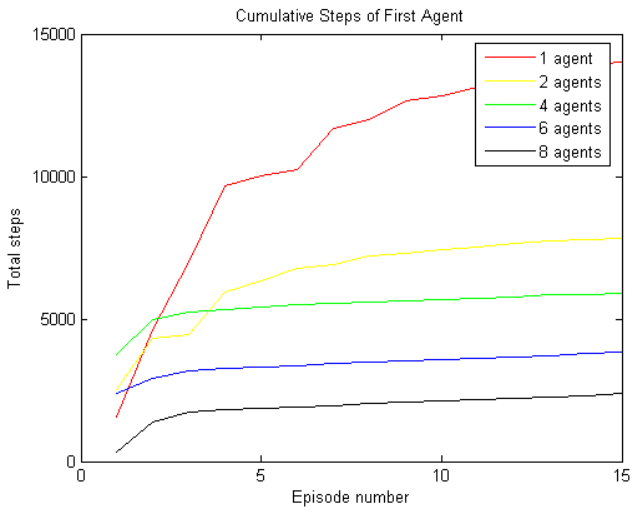


Fig. 4. Using multiple agents increases the learning rate

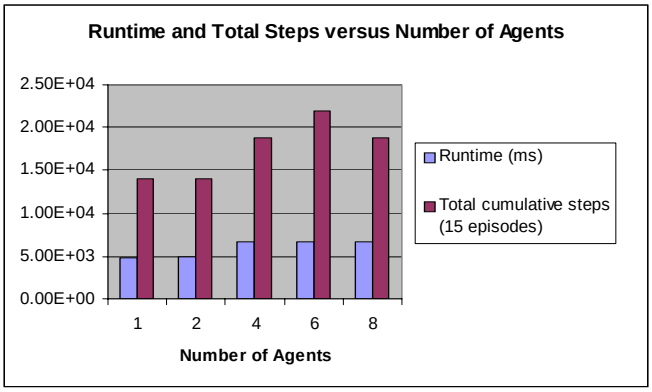


Fig. 5. Runtime and total steps increase as the number of agents increases

## IV. HARDWARE DESIGN OF MULTI-AGENT SARSA($\lambda$)

Adapting Sarsa ($\lambda$) for a custom hardware implementation opens a number of possible optimizations parameters, however it also introduces several potential pitfalls. First, concerns pertaining to a single agent processing element are addressed. Subsequently, a hardware system with multiple agents is discussed.

### A. Single-Agent in Hardware

It is highly desirable to formulate the learning algorithm as an integer-only problem, as custom hardware typically suffers from degraded performance when operating with floating point values. However, integers have a much narrower dynamic range, so that care must be taken to evaluate whether the problem is tractable given the integer representation and discount factor. As an example, consider a 32-bit integer with a discount factor of $\gamma = 0.5$. The maximum value of the grid world problem is the 32-bit reward for the state-action resulting in the goal state. Each step leading up to the goal approaches half the value of its successor state, so that states located further than 32 steps away from the goal necesarily have a value of zero. With these example parameters, an optimal solution of greater than 32 steps cannot be reliably found. Keeping this concern in mind, the 32-bit integers and $\gamma = 0.75$ are used in evaluating the $15 \times 15$ grid world task.

A second concern, given the limited representation range of integers, is the exhaustive state-action space sweep dictated by the algorithm: it is reduced to the most recently visited states. Since the eligibility trace updates are scaled by powers of the combined factor $\gamma\lambda$, the horizon is significantly less than that of the state-action values. For each time step, $k$, past a state-action, the associated eligibility trace is $(\gamma\lambda)^k e(s,a)$. A buffer of the most recently visited state-actions can be maintained, and the values corresponding to those state-actions are the only values to be examined and updated. For this problem, a range of 12 states is examined. These updates are independent of one another, so they can be performed in parallel using hardware, given that the necessary memory bandwidth is available.

Another optimization technique that can be applied has to do with bit shifts that can be substituted for multiplications, if the various factors are chosen carefully. The chosen factors of 0.5 and 0.75 can be represented as a bit shift or two bits shifts and an addition, respectively. For the eligibility trace updates of $(\gamma\lambda)^k$, the operations can be approximated as shifts where the shift amount for each state in the buffer is pre-calculated and stored.

Finally, a speed-up can be obtained by locally buffering the state-action values associated with the eligibility trace updates; this results in an "efficient" update method rather than the "precise" update method dictated by the algorithm. Rather than receiving the relevant state-action values from memory for each eligibility trace update, only the local buffer need be consulted. When a new step is taken, the existing states in the buffer can be shifted and the new value can be added. The value that moves out of the buffer can be written back to memory. To the best knowledge of the authors, this is a novel approach to dealing with the computationally-intensive eligibility traces, with the side effect of delayed updates to the state-action value table.

In an $N$-stage buffer, the exact update scheme would perform $N$ reads and $N$ writes. In comparison, the efficient scheme would only perform one of each. For a single-agent, this efficient update scheme is similar to using replacement traces with a delayed replacement. Yet, for many multiple agents, this scheme has limitations since agents often store out-of-date values in their buffers.

### B. Multi-Agent in Hardware

With true hardware parallelism, the increased processing time required for multiple agents can be minimized. The greatest challenge in implementing multiple agents is their collaboration - namely exchanging information in a timely manner. A shared-memory SMP architecture is one way to address the issue, and to the knowledge of the authors, this is the first use of SMP in reinforcement learning. As explored

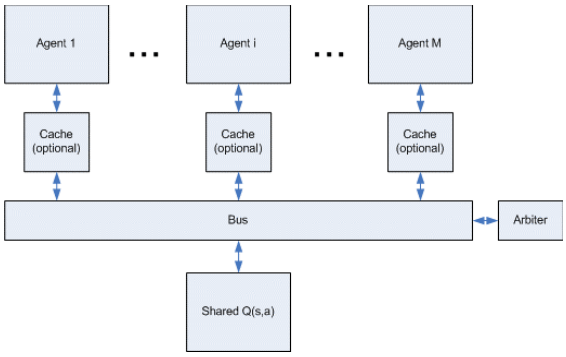


Fig. 6. An SMP architecture provides shared memory for parallel agents

in [2], multiple agents are most efficient when they quickly update their value estimates. This is a question of processing speed and bandwidth. Specialized hardware addresses both of these issues by optimizing the logic functions and providing tight integration between the logic and memory. Scalability might be a concern with an SMP architecture, but [2] also illustrates the diminishing rewards of adding additional agents due to overlap in their experience. Thus for a relatively small number of agents, SMP seems to be a promising approach. An additional benefit of using SMP is its adaptability - each agent is allowed to independently explore the entire problem space according to its own policy while the architecture itself ensures that their shared memory space is properly updated. In contrast, the strict partitioning scheme in [3] requires a master/slave relationship in which each agent is confined to a particular region of the problem space.

Using the SMP paradigm, each agent is viewed as a processing element (PE) and communicates with a shared memory over a shared bus. This architecture is shown in figure 5. The bus arbitrates between the requests, serializing the memory accesses and allowing the single-port memory to be accessed by all agents. To improve performance and reduce the required memory bandwidth, caches reside between the agents and the bus. Caches can accelerate performance by offering locally-accessible data, however they can also add latency and complexity when required data is not available locally. The relative importance of the caches and their optimal sizes is problem dependent on the problem space and is explored in later sections.

In the constructed model, the agents are single-cycle processing elements that stall while waiting for memory. The 32-bit wide bus arbitrates memory accesses based on a priority queue in which the first agent (used for performance measurements) has the highest priority. Since the intended target device for a future implementation is an FPGA, a 100 MHz clock rate is selected, and synchronous memories with 1 cycle read delays imitate FPGA block RAMs. The caches, when instantiated, are directly mapped and make use of one-word blocks. Cache coherency is not enforced, but as will be shown later this has little effect on accuracy or performance.

Two other implementation details have been avoided since they are outside the scope of this paper. First, for the $\varepsilon$-greedy policy followed by each agent, a random number generator is needed. Parallel random number generation is relatively straightforward and could use parallel uniquely-seeded linear feedback shift registers, as well as other mechanisms. This method is fast and efficient and would scale easily with multiple agents. Second, the means for interacting with a model or environment are not discussed. For a shared model, a similar bus and shared-memory architecture could be implemented with its own bus and memory. This would involve mirroring the presented architecture and adapting it for the application. For the sake of modeling the multi-agent system, each agent is assumed to have its own unique, though identical, model.

## V. Performance Evaluation

To examine the merits of this system, a cycle-accurate software model was constructed using SystemC [5]. The parameterized model facilitates evaluation of the system's performance with any number of agents, exact or efficient updates, cache or no cache, and various sizes of cache. SystemC was chosen for its relative ease of use (C-like hardware modeling) and its recognized place in system-level architectural exploration methodologies [6]. SystemC's performance is up to 10,000x that of traditional hardware design simulation. [7].

The multi-agent system was tested under a number of different configurations, and in each the wall clock time was summed over 15 episodes. The configurations, and their corresponding simulated hardware runtimes, are shown in figure 7. As expected, the hardware is several orders of magnitude lower than that of the softare: several milliseconds versus several seconds. The hardware offers a massive performance improvement due to its customization, and it can be seen that the hardware gives a performance increase in both the single-agent and multi-agent systems. However, the performance boost is not completely accurate, since although both implementations used the same algorithm, the software was run in the high-level Matlab environment.

In contrast with the previous software-based runtimes, the actual execution time generally *decreases* as more agents are added to the multi-agent system. Experience is shared among the agents in both the software and hardware environments, but in the hardware environment the agents gain their individual experience in parallel. As outlined in [2], the experience gained overlaps, leading to less than linear improvements in learning rate. Additionally, adding additional agents to the system increases the traffic on the shared bus, leading to contentions that delay updates/reads from the shared state-action table. Slight discrepancies in these patterns, such as the abnormally-low exact update time for two agents and 8 words of cache, are likely due to the problem space of the particular task.

The efficient update method, when used with a single agent, produces indistinguishable results compared with the exact update method, and it offers a clear performance boost by running 2-4x faster. The benefit is greatest in the single agent case because the skipped state-action updates result only in fewer memory operations and therefore save several cycles.

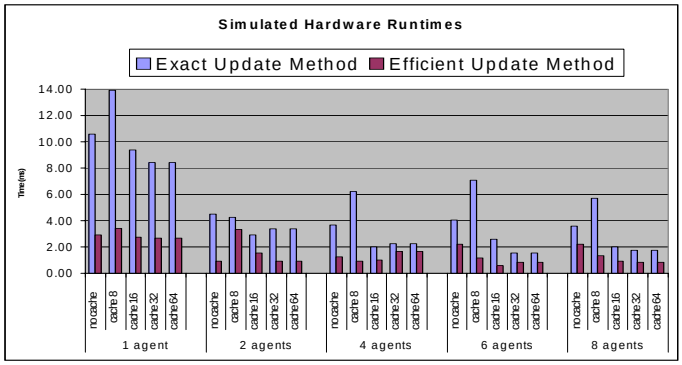


Fig. 7. The simulated hardware runtimes show a clear performance benefit with parallel agents, and furthermore the optimizations discussed provide a significant performance boost

Once additional agents are added to the system, however, the benefit of the efficient update is lessened since skipped state-action updates also result in delayed collaboration and the discarding of some agents' updates. The system using efficient updates still converges to the optimal solution, leading to the conclusion that the benefit of having multiple agents exploring the state-action space is not greatly impacted by slight inconsistencies in their state-action value functions, as long as they are interacting with the same underlying environment model. The downsides of te efficient method are empirically low, being more than offset by the reduction in memory accesses.

It can be observed that adding some amount of cache to the system usually decreases runtime. For the single-agent case, the benefit of adding the 8 block cache does not offset its added latency, but increasing the cache helps for the exact update case. The single-agent efficient update does not benefit from additional cache because it is not performing many memory accesses. Interestingly, the 2, 4, 6, and 8 agent cases show a shifting sweet-spot for the cache size. For exact updates, the best performance comes from the 16 block, 16 block, 32 block, and 64 block cache configurations, respectively. For the efficient updates, the best performance comes from the no cache, 8 block, 16 block, and 32 block cases, respectively. These trends reflect the increasing stress on the shared memory bus as the number of agents increases. Since the ideal solution for this problem is less than 40 steps, the 64 word cache is large enough to hold nearly all of the relevant values, such that runtimes with larger caches are redundant. Larger caches are also made unnecessary by the diminishing returns from adding additional agents.

## VI. Conclusions and Future Work

This paper presented a hardware-based architecture for realizing a high-speed, multi-agent reinforcement learning system. While multiple software agents can work together to learn from fewer episodes, parallel processing in hardware is needed to increase the real-time learning rate. Several design considerations and optimization techniques for implementing Sarsa($\lambda$) agents in hardware were presented herein, and the results of cycle-accurate modeling of an SMP architecture were discussed, accentuating the merits of the proposed approach. The model verifies that hardware can achieve significant performance gains over its software counterpart, and these gains are bounded since additional synchronization is required for each additional agent.

The increase in performance can be critical when learning in fast-changing environments. There, planning with models can be useful only when the models can be executed much more quickly than the environment can respond, and this may only be possible with a parallel hardware implementation.

While the performance gains possible with parallel hardware are tantalizing, a number of issues remain to be addressed. Of chief concern is this model's reliance on integer data types, which limits the size of the problem being addressed. Some capability can be gained by increasing the number of bits allocated to the value function and by tweaking the parameters and their implementation, however the use of integers severely limits the hardware flexibility when compared to floating point arithmetic operations.

The Sarsa($\lambda$) method discussed herein uses a tabular value function, and the tabular function also leads to intractability problems as the size of the state space increases. To combat this problem, further study is required as to how functional approximation methods can be efficiently represented in hardware to effectively manage large state spaces. Additionally, the SMP architecture modeled herein did not make use of cache coherency. Although the impact of coherency on this application was minimal, other problem sets or learning algorithms might not show the same robustness. Cache coherency in an SMP environment is well studied, and this hardware model can be extended to evaluate its impact.

## VII. Acknowledgement

The authors would like to thank Zhenzhen Liu for her many insightful suggestions and comments regarding this work. This work has been partially supported by the Woodrow W. Everett, Jr. SCEEE Development Fund in cooperation with the Southeastern Association of Electrical Engineering Department Heads.

## References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge MA, MIT Press, 1998.

[2] R. M. Kretchmar, "Reinforcement learning algorithms for homogenous multi-agent systems," *Workshop on Agent and Swarm Programming (WASP'03)*, 2003.

[3] A. Printista, M. Errecalde, and C. Montoya, "A parallel implementation of q-learning based on communication with cache," *Journal of Computer Science and Tenchnology*, vol. 6, 2002.

[4] G. Laurent and E. Piat, "Parallel q-learning for a block-pushing problem," pp. 286–291, 2001.

[5] "Systemc version 2.0 user's guide," 2002. Available http://www.systemc.org.

[6] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, and et al., "A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms," 2004.

[7] T. Rissa, A. Donlin, and W. Luk, "Evaluation of systemc modelling of reconfigurable embedded systems," vol. 3, 2005.