Bandwidth-Sensitivity-Aware Arbitration for FPGAs

Lu Hao and Greg Stitt, *Member, IEEE*

***Abstract*—Field-programmable gate arrays (FPGAs) commonly implement massively parallel circuits that require significant memory bandwidth. Due to I/O and memory limitations, parallel tasks often share bandwidth via arbitration, whose efficiency is critical to ensure parallelism is not wasted. In this letter, we introduce a bandwidth-sensitivity-aware heuristic for arbitration that analyzes the effect of memory bandwidth on performance for each application task, and then accordingly allocates bandwidth to minimize execution time. When compared to round robin (RR) arbitration, application speedups as high as 6.5× are achieved.**

***Index Terms*—Arbitration, field-programmable gate arrays (FPGAs), memory bandwidth, reconfigurable computing.**

I. Introduction

FIELD-PROGRAMMABLE gate arrays (FPGAs) are commonly used in embedded systems and high-performance computing to speed up applications [4], [6] via pipelines requiring significant memory bandwidth. For many FPGA applications, insufficient bandwidth is the main bottleneck. To address this, FPGA platforms use multiple external memories [3]. However, adding more memory is often not feasible due to I/O limitations, which are worsened by rapidly increasing FPGA resources that demand even more bandwidth.

As a result, FPGA tasks share memory bandwidth via arbitration. Although arbitration has been widely studied [2], [7], and [9]–[11], the previous work mainly focused on priority assignments that do not consider application-specific behaviors. The reconfigurability of FPGAs enables application-specialized arbitration.

In this letter, we introduce the *bandwidth-sensitivity-aware* (BSA) heuristic for application-specialized arbitration. One novel characteristic of BSA is the consideration of *bandwidth sensitivity* of application tasks, which defines how task execution time is affected by memory bandwidth. Based on these sensitivities, BSA effectively allocates memory bandwidth by providing highly sensitive tasks (i.e., pipelines) with more bandwidth than tasks that are less sensitive to bandwidth (i.e., control). For example, Fig. 1 compares round robin (RR) arbitration with BSA for an application consisting of three independent tasks. As shown, Task 1 is highly sensitive to bandwidth, while Tasks 2 and 3 are much less sensitive to it. For this example, RR arbitration would simply allocate 33% of the memory bandwidth to all tasks. However, BSA considers the higher sensitivity of Task 1 and allocates 80% of the bandwidth, while allocating 10% to both Task 2 and Task 3. For this example, the execution time is the maximum time of all three tasks, for which BSA achieves a speedup of approximately 2×.

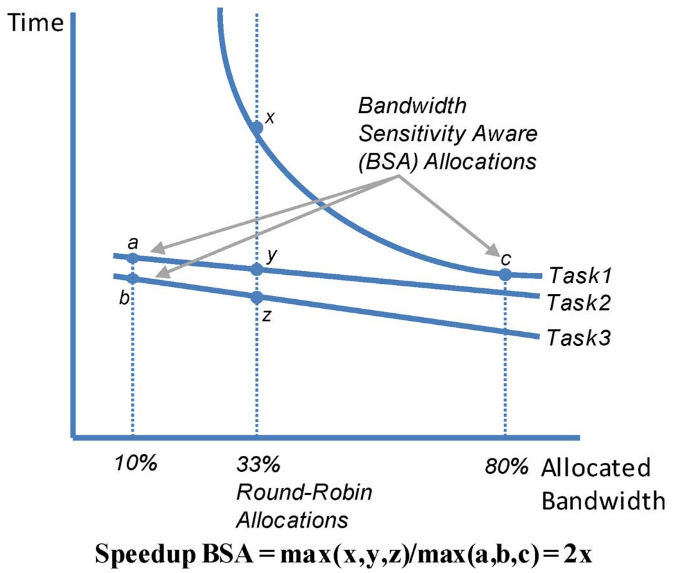


Fig. 1. A comparison of round robin and bandwidth-sensitivity-aware arbitration for three parallel tasks (total execution time is the maximum of all tasks).

We evaluate BSA using embedded applications, showing application speedups up to 3.9× compared to RR. We also simulate synthetic applications under varying conditions and use cases, showing speedups up to 6.5×.

II. Previous Work

Previous studies focused on arbitration for shared resources such as memory and buses. Much of that work statically assigns access priorities for each task, using algorithms such as static priority, TDM, Lottery, and RR [10], [11]. Arbitration is also common in real-time systems, which focus on meeting hard real-time constraints (i.e., RT_Lottery [2] and Warning-zone [9]). Other approaches focus on soft requirements such as slack-based bus arbitration for streaming multimedia processors [7]. These earlier approaches assume that all tasks are equally sensitive to bandwidth from the shared resource. In addition, they generally assume that bandwidth has a linear effect on the performance of tasks. Our work instead considers different bandwidth sensitivities, which can vary significantly between tasks in FPGA applications. For example, for a pipelined

Manuscript received May 18, 2012; accepted June 21, 2012. Date of publication July 18, 2012; date of current version September 14, 2012. This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant EEC-0642422. This manuscript was recommended for publication by J. Ayala.

L. Hao is with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: lucia.hao@gmail.com).

G. Stitt is with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA, and also with the NSF Center for High-Performance Reconfigurable Computing, Gainesville, FL 32611 USA (e-mail: gstitt@ece.ufl.edu).

Color versions of one or more of the figures in this letter are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/LES.2012.2209397

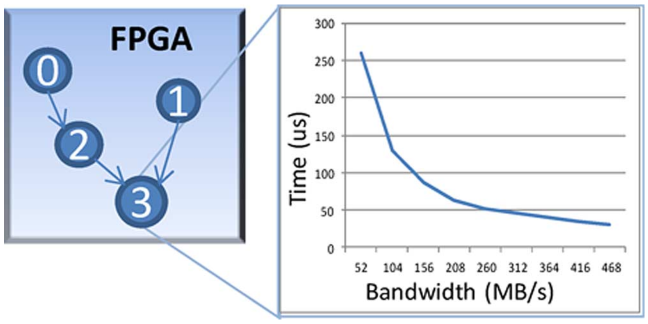


Fig. 2. An example application task graph with time–bandwidth curves.

task, the throughput is proportional to the allocated bandwidth, up to some maximum bandwidth. The resulting execution time is the total amount of data divided by the throughput, resulting in a nonlinear curve. By considering bandwidth sensitivity, BSA is able to specialize arbitration to different applications based on these curves. Furthermore, BSA supports specialization for different phases of execution.

Yang *et al.* [13] scheduled multiple tasks on a single processor to meet task deadlines while minimizing energy and considered an energy-time Pareto curve for each task, which is analogous to a time–bandwidth curve in application-specialized arbitration. To solve the scheduling problem, Yang *et al.* mapped their problem to the multiple-choice knapsack problem and used a greedy heuristic. Because the heuristic from [13] could potentially be used for application-specialized arbitration, we compare it with BSA and show performance improvements averaging 1.8×.

```
01 MAIN FUNCTION
02 Solution s = Randomly select bandwidth priority (BP) for all tasks;
03 for $N_t$ initial solutions
04     Time[s] = GetSolutionTime(s);
05     s = Randomly change the BP of one task;
06 endfor
07 Compute temperatures $T_{start}$ and $T_{end}$ according to all Time[] values;
08 $T_{current} = T_{start}$; $s_{current} = s$;
09 while $T_{current} > T_{end}$
10     s = Randomly change the BP of one task;
11     Time[s] = GetSolutionTime(s);
12     if Time[s] < Time[$s_{current}$] then
13         $s_{current} = s$;
14     else if exp((Time[s] – Time[$s_{current}$]) / $T_{current}$) > random() then
15         $s_{current} = s$;
16     endif
17     if reached maximum number of solutions at $T_{current}$
18         $T_{current} = \alpha T_{current}$;
19     endif
20 endwhile
21 return $s_{current}$;
22 function GetSolutionTime(Solution s)
23     time = 0;
24     do
25         for each task t that has not executed
26             if predecessors of t have executed
27                 Identify t as a candidate;
28             endif
29         endfor
30         for i = 0 to number of candidates
31             Bandwidth[i] = Total bandwidth × BP[i] / ∑BP;
32         endfor
33         time += time until first candidate completes;
34     until all tasks have executed;
35     return time;
36 endfunction
```

Fig. 3. Overview of BSA heuristic.

III. Problem Definition

The input to the application-specialized arbitration problem is an application task graph, where each task executes whenever inputs are available and completes after a number of inputs. Each task has a bandwidth sensitivity represented by a time–bandwidth curve that shows task execution time for different arbitration-allocated bandwidths.

Fig. 2 shows an example time–bandwidth curve for Task 3. Although in this letter we assume the time–bandwidth curves are an input to the problem, such curves could be determined in a variety of ways. For example, high-level synthesis could estimate the curves based on memory-access patterns. Alternatively, device vendors could benchmark IP cores. In our experiments, we physically measured curves for tasks.

With these definitions, we define the application-specialized arbitration problem as follows. Given an application in the form of a task graph and a set of time–bandwidth curves for each task, create an arbitration policy that allocates a specific bandwidth to each task at all points during execution, such that the total execution time of the application is minimized.

IV. Bandwidth-Sensitivity-Aware Heuristic

To consider bandwidth sensitivity, we initially considered existing greedy heuristics [13]. However, there was no clear greedy strategy because to optimally allocate bandwidth, the heuristic needs the number of concurrent tasks at any point in time, but to determine the number of concurrent tasks, the heuristic needs to have already allocated bandwidth. To deal with this circular dependency, BSA uses simulated annealing.

Fig. 3 illustrates BSA, which creates an initial solution (Line 2) by randomly assigning a bandwidth priority (*BP*) to each task (using cstdlib rand()) that BSA later uses to allocate bandwidth. Although random priorities may seem counter-intuitive, BSA uses this strategy due to the previously mentioned circular dependency between concurrency and bandwidth. Next, BSA calculates the application execution time based on this solution (Line 4), as discussed in the following paragraph. BSA then explores N_t (the number of tasks) other initial solutions (Lines 3–6), which are used later to calculate starting and ending temperatures (Line 7). Each one of these initial solutions randomly selects one task and randomly changes its *BP*. The remainder of BSA iterates over neighboring solutions consisting of a randomly changed *BP* for one task. If the solution is better than the current best, BSA always accepts it (Lines 12–13); otherwise, BSA accepts it with a probability (Lines 14–15) that decreases based on temperature. For each temperature, BSA generates a number of solutions and then updates the temperature using a cooling schedule (Lines 17–18), discussed later. The anneal ends when reaching the final temperature T_{end}.

To calculate execution time for a solution (Lines 22–36), BSA selects execution candidates (i.e., tasks whose predecessors have executed) and then allocates bandwidth for each candidate by multiplying the total memory bandwidth by the candidate's BP, divided by the sum of all candidate BP values

(Lines 30–32). BSA then determines the first candidate task to complete and increases the time based on that task's curve (Line 33). The algorithm iteratively repeats these steps, referred to as a *round*, until all tasks are complete.

To optimize simulated annealing, we used existing strategies [8]. BSA sets the starting temperature $\mathrm{T}_{\mathrm{start}}$ (Line 7) to accept any solution by using either 20 times the standard deviation or 10 times the average of the first N_{t} solutions, whichever is larger. At each temperature, BSA evaluates $30 \times (\mathrm{N}_{\mathrm{t}})^{1.33}$ solutions, as suggested in previous work [1], which we empirically adjusted by $3\times$.

The authors in [8] recommend keeping the acceptance ratio near 0.44 for as long as possible. To achieve this goal, BSA adapts the cooling schedule from [1], $\mathrm{T}_{\mathrm{Current}} = \alpha \mathrm{T}_{\mathrm{current}}$, where α ranges from 0.1 for high acceptance ratios values to between 0.8 and 0.95 for smaller ratios.

We empirically evaluated different T_{end} values and chose the average time for the first neighboring N_{t} solutions divided by 50 000. A different set of task graphs might benefit from a different T_{end}. Although a complete analysis of the tradeoffs of different T_{end} values is outside the scope of the letter, our experiments also showed that by gradually reducing the number of solutions by 10% at each temperature, the total heuristic time was reduced by 55%, whereas the speedup was only reduced 0–5% for streaming applications. For non-streaming applications, the total heuristic time was reduced by 27%, whereas the speedup was only reduced 0–0.6%.

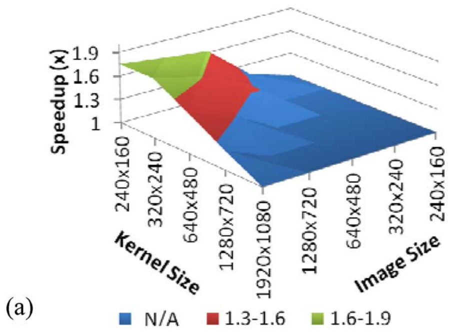
V. Experiments

We implemented BSA in C using approximately 4000 lines of code. Bandwidth allocation was determined offline using BSA. We also created a VHDL arbiter that uses a time– division multiplexing method, where the size of each time slot corresponds to the priority determined by BSA for each round. We compared BSA with RR, in addition to the heuristic from [13], which we refer to as the Yang heuristic. To perform this comparison, we regenerated Yang's work and mapped it onto the arbitration problem. We treated every round as an instance of the multiple-choice knapsack problem, and then used Yang's heuristic to solve each round. During each round, Yang uses the time value of the leftmost point of each curve as the initial bandwidth allocation and then uses a greedy heuristic to minimize the sum of times for all candidates. Yang's work finds a solution for each round and adds them together without considering dependencies between rounds, whereas BSA instead finds a solution for the entire task graph. Therefore, in general, Yang's work cannot guarantee as efficient a solution for the entire application because of task dependencies.

To evaluate BSA, we tested streaming and nonstreaming applications, due to different arbitration requirements. The former represents pipelined applications on FPGAs; the latter represents unpipelined tasks. All results include both computation and communication times.

A. Application Case Studies

We evaluated BSA using three applications: 1) sum of absolute difference (SAD) image comparison; 2) 2D convolution; and 3) an MPEG-2 encoder/decoder. We determined each task's time–bandwidth curve using physical measurements and then determined application speedup via simulation of the task graph, using each type of arbitration.



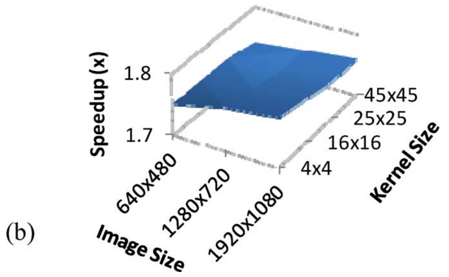


Fig. 4. BSA application speedup compared to Yang for (a) SAD and (b) 2D convolution input sizes.

SAD measures similarity between a region of an image and a different kernel image. We extracted the task graph from [5], a streaming implementation. To measure time–bandwidth curves, we provided tasks with input data from SODIMMs on a GiDEL PROCStar III with an Altera Stratix III E260. We stored this data in first-in-first-out (FIFO) buffers where we restricted the bandwidth and measured the corresponding task execution time. We varied image sizes and kernel sizes from 240×160 to 1920×1080, respectively. For this version of SAD, the algorithm slides a feature-sized input window over the entire input image, regardless of the number of matches. Fig. 4(a) shows the application speedup of BSA compared to Yang. The speedup was highest ($1.75\times$) when the image and kernel size had the biggest differentials (i.e., 1920×1080 image with 240×160 kernel), and was $1\times$ when the kernel and the image had the same sizes. These trends were caused by higher bandwidth sensitivities for larger differences in image and kernel sizes. When using a large image and a small kernel, the task reading the image is more sensitive, and vice versa. For similarly sized images and kernels, the task sensitivities are similar, which limits BSA improvements. Speedup compared to RR was higher, peaking at $2\times$.

For 2D convolution, as shown in Fig. 4(b), we extracted the task graph and time–bandwidth curves for 2D convolution from the implementation described in [5], which is also a streaming implementation. Task graphs and curves were built the same as they were with SAD. The image sizes were varied from 640×480 to 1920×1080, and kernel sizes were varied from 4×4 to 45×45. BSA speedup remained around $1.75\times$ compared to Yang, again due to differences in bandwidth sensitivities between image sizes and kernel sizes. Speedup compared to RR was approximately $2\times$.

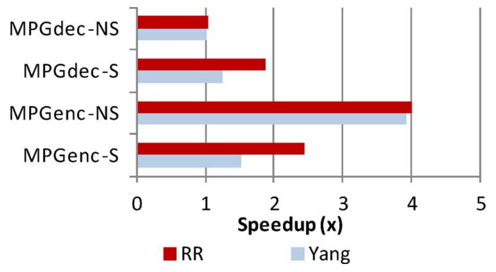


Fig. 5. BSA application speedup for MPEG encoding/decoding for streaming (S) and nonstreaming (NS) implementations compared to Yang and RR.

We evaluated MPEG encoding/decoding both with and without streaming because both models are common. The MPEG-2 encoder/decoder was from the ALPBench benchmark suite. We used Simics and CETA to extract task graphs and to trace data transfer patterns between tasks. Fig. 5 shows the speedup comparison for a video with 128 × 128 resolution. For the streaming version, BSA provided speedups of 1.3× and 1.5× compared to Yang's work, and 1.9× and 2.4× compared to RR. For nonstreaming results, BSA performance was much better for the encoder (4× compared to RR) due to more tasks and highly varied curves. For the decoder, BSA speedup was only 1.01× compared to Yang due to few tasks and little variance among curves.

B. Simulated Arbitration Results

To complement the previous case studies, we use synthetic applications with widely varying, randomly generated task graphs and time–bandwidth curves. To stress test BSA, we did not include tasks that are completely independent of memory bandwidth (i.e., control tasks). The nonstreaming synthetic applications were communication bound, with approximately 95% of execution time spent communicating.

For streaming applications, we varied the tasks from 3 to 19, which we found to be representative of previous FPGA applications [5], [12]. The solid lines in Fig. 6 show the average speedup obtained by BSA compared to both Yang (1.3×) and RR (4.4×). Compared to RR, BSA performance increased rapidly with increasing tasks due to more parallelism and curve variance exploration, whereas a stable speedup was achieved compared to Yang.

For nonstreaming applications, we varied the tasks from 20 to 180, represented by the dashed lines in Fig. 6. We evaluated larger numbers of nonstreaming tasks because unpipelined tasks tend to use less area on FPGAs. The average BSA speedup was 3.5× compared to RR and 2.1× compared to Yang. Speedup gradually decreased because the larger task graphs had longer paths through the task graph without a similar increase in parallelism.

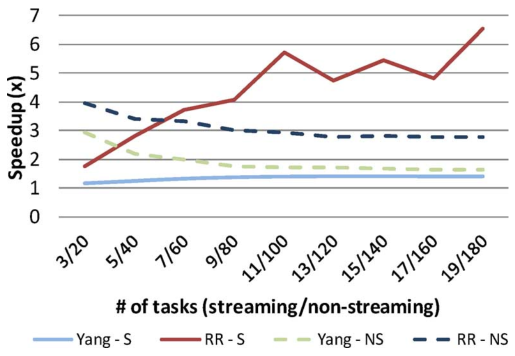


Fig. 6. BSA application speedup for streaming (S) and nonstreaming (NS) applications compared to Yang and RR.

VI. Conclusion

In this letter, we introduced the BSA arbitration heuristic, which considers bandwidth sensitivity of application tasks to reduce execution time. Compared to the more commonly used RR arbitration, BSA achieved application speedups as high as 6.5×. We also adapted an existing scheduling heuristic as a potential arbitration solution and showed that BSA provided a speedup averaging 1.8×.

References

[1] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. Int. Workshop Field Program. Logic Appl.*, Aug. 1997, pp. 213–222.

[2] C.-H. Chen, G.-W. Lee, J.-D. Huang, and J.-Y. Jou, "A real-time and bandwidth guaranteed arbitration algorithm for SoC bus communication," in *Proc. ASP-DAC*, 2006, pp. 600–605.

[3] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Comput. Surveys*, vol. 34, no. 2, pp. 171–210, Jun. 2002.

[4] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[5] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," *FPGA*, pp. 47–56, 2012.

[6] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors," *FPGA*, pp. 162–170, 2004.

[7] M. Jun, K. Bang, H.-J. Lee, N. Chang, and E.-Y. Chung, "Slack-based bus arbitration scheme for soft real-time constrained embedded systems," in *Proc. ASP-DAC*, 2007, pp. 159–164.

[8] J. Lam and J. M. Delosme, "Performance of a new annealing schedule," in *Proc. DAC*, 1988, pp. 306–311.

[9] H.-K. Peng and Y.-L. Lin, "An optimal warning-zone-length assignment algorithm for real-time and multiple-QoS on-chip bus arbitration," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 4, pp. 1–39, Apr. 2010.

[10] F. Poletti, D. Bertozzi, L. Benini, and A. Bogliolo, "Performance analysis of arbitration policies for SoC communication architectures," *J. Des. Autom. Embed. Syst.*, pp. 618–621, 2003.

[11] C. H. Pyoun, C. H. Lin, H. S. Kim, and J. W. Chong, "The efficient bus arbitration scheme in Soc environment," in *Proc. Int. Workshop System-on-Chip Real-Time Appl.*, 2003, pp. 311–315.

[12] V. M. Tuan, N. Katsura, H. Matsutani, and H. Amano, "Evaluation of a multicore reconfigurable architecture with variable core sizes," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–8.

[13] P. Yang and F. Catthoor, "Pareto-optimization-based run-time task scheduling for embedded systems," in *Proc. CODES+ISSS*, 2003, pp. 120–125.