Constraint driven Scheduling of fine-grained C concurrency for Reconfigurable Hardware

1 Introduction

High Level Synthesis(HLS) is a process of mapping high level(C-like programs) models of hardware to low level concrete (HLD-like programs) ones; which is then synthesized as hardwares like FPGAs. HLS tools are used widespread in recent times due to the high demand for specialized hardware than can perform better for specific Machine Learning applications.

The past decades have also progressed in better understanding how to better utilize multicore machines for several applications. Specifically, the use of fine grained concurrent features (eg: Load/Store buffers, speculation,etc.); whose descriptions are termed as relaxed memory consistency models.

Our focus is in the marriage of scheduling in HLS and such fine grained concurrent programs that can be used to synthesize hardware. Scheduling is a phase in HLS wherein programs (models) are mapped to hardware clock cycles. The schedule determines how many clock cycles would it take for a computation to be done on the hardware synthesized.

We specifically look at the impact on scheduling such programs given a constraint on number of hardware resources available to map concurrent processes"threads". Our contribution in this direction three fold:

- Investigate the impact in scheduling such programs given thread resource constraint.
- Implement pre-scheduling optimization to improve the overall schedule.
- Add a global-analysis to improve the optimization to get a more efficient schedule.

2 Related Work

[1] show how to map software threads to parallel hardware for FPGAs. They do this for programs that utilize pthreads and OpenMP constructs. However, their focus was mainly on lock based concurrency in C. While they do note in their experiments that having a constraint on resources to map threads does effect the schedule(more cycles), they do not investigate if any optimization pre-scheduling can be done to improve this.

[7] and [8] address the scheduling problem for synthesis of concurrent programs utilizing fine grained concurrency elements of C. They show that while using atomics primitives provided by C, incorrect schedules can be obtained. They remedy this by adding additional ordering constraints among memory accesses. They also implement pipelined scheduling for such programs.

[2] and [3] show the impact/effect of compiler optimizations on HLS for Reconfigurable hardware like FPGAs. Specifically, they show how scheduling(termed as latency) has a major impact due the optimization passes of the compiler, as well as the order in which we perform it. But their work does not address this impact on concurrent programs(let alone those using fine grained Concurrency) synthesized to hardware.

[9] shows that programs utilizing fine grained concurrency of C may not get the benefit of some optimizations as they are rendered unsafe in a concurrent context.

[5] try to identify which memory accesses can be safely reordered in a weakly consistent C program targeted for hardware synthesis. Their goal is again to improve the overall schedule of the program, and in turn the hardware that is synthesized for it. [6] is a follow up to this work describing a global analysis to more aggressively perform such reordering. This reordering resulted in a better schedule overall.

Previous works in the line of synthesizing weakly consistent C programs assume that every thread can be mapped to a unique hardware accelerator during synthesis. They do not investigate the effect of constraining the number of available accelerators, and whether any optimization can be done for this to improve the schedule.

3 Methodology

In the presence is resource constraint for mapping threads to hardware, more than one thread's code will have to be run by the same hardware resource. This brings about an interleaving behavior between those threads, which is coupled by "context switching" to ensure each thread's

Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

code is executed. This ”context switching” will cost more cycles, and overall would lead to an inefficient schedule.

Sequentialization is a compiler transformation that merges two threads by placing one thread's code after another. [4] have thankfully recorded that sequentialization can be done safely for weakly consistent C programs (given the recent C11 memory consistency model). We propose to augment this optimization in pre-scheduling stage given constraints on hardware resources for threads.

Such a transformation for our purpose would at first sight, save the context switching overhead when it comes to scheduling more than one concurrent thread to a hardware resource. We plan on adding this transformation before scheduling our programs to address the ”context switching” overhead.

Traditional lock based concurrency would only benefit from the saving of ”context switch” cycles when it comes to scheduling shared memory accesses. But we note, from previous work's observation, that our transformation would benefit more if we have fine grained concurrent elements. Merging two thread's code can expose more memory accesses that are independent and can be scheduled in the same clock cycle. Additionally, taking motivation from previous work, we also would like to identify a global analysis to better perform this optimization.

In our current work, we will only show how this transformation in unison with reordering improves the schedule. But it is important to note that several other optimizations can also further give us a better scheduled circuit. We leave this analysis as part of future work.

For our contribution, we use our existing code base which synthesizes a subset of C like sequential program to hardware level VHDL language. We extend our code base to incorporate the notion of threads, followed by augumenting it with weakly consistent memory accesses from previous work. We then place a constraint on the resource count for threads and observe the schedule that results due to previous work. We then augment our naive version of transformation that performs sequentialization before scheduling. We also then add a pre-analysis step for this transformation that results in a more efficient merging of threads which result in a better schedule. We analyze the impact of our two approaches to previous work on the resultant schedule(s) obtained. For this, we utilize the benchmarks used in previous work, along with adding our own custom made programs which can expose the importance of the pre-analysis step. In summary, we do the following:

- Encode previous work's contribution to our existing code base.
- Constrain the amount of hardware resources available to map threads.
- Augment sequentialization transformation, that chooses random pairs of threads to merge, until we can uniquely map each thread to a resource.
- Add a global-analysis step to better identify pairs of threads to sequentialize.
- Observe the outcome of the resulting schedule on lock free algorithms of message-passing and producer-consumer problems.
- Observe the outcome of the resulting schedule on custom written lock free programs.

Note that our current contributions are only restricted upto the scheduling stage on HLS. As part of our future work, we would like to incorporate our contribution to existing HLS tools like LegUp, that can synthesize concurrent models like ours to hardware.

4 Timeline

- Week 1,2: Add AST node Thread and write sample programs to test scheduling of memory accesses.
- Week 3: Add types for each memory access, test benchmark program naive scheduling (incorrect one).
- Week 5: Augment previous work addition to scheduling constraints, test benchmark programs.
- Week 6: Augment thread resource constraint and merging transformation, test benchmark program schedules.
- Week 7,8: Add algorithm that takes program and gives out set of threads to inline, test benchmark program schedules.
- Week 9: Report results.

References

[1] Jongsok Choi, Stephen Dean Brown, and Jason Helge Anderson. 2013. From software threads to parallel hardware in high-level synthesis for FPGAs. In *2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013*. IEEE, 270–277. https://doi.org/10.1109/FPT.2013.6718365

[2] Jason Cong, Bin Liu, Raghu Prabhakar, and Peng Zhang. 2012. A Study on the Impact of Compiler Optimizations on High-Level Synthesis. In *Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers (Lecture Notes in Computer Science)*, Hironori Kasahara and Keiji Kimura (Eds.), Vol. 7760. Springer, 143–157. https://doi.org/10.1007/978-3-642-37658-0_10

[3] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Dean Brown, and Jason Helge Anderson. 2013. The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs. In *21st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2013, Seattle, WA, USA, April 28-30, 2013*. IEEE Computer Society, 89–96. https://doi.org/10.1109/FCCM.2013.50

[4] Evgenii Moiseenko, Anton Podkopaev, and Dmitrij V. Koznov. 2021. A Survey of Programming Language Memory Models. *Program. Comput. Softw.* 47, 6 (2021), 439–456. https://doi.org/10.1134/S0361768821060050

[5] Nadesh Ramanathan, George A. Constantinides, and John Wickerson. 2018. Concurrency-Aware Thread Scheduling for High-Level Synthesis. In *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. IEEE Computer Society, 101–108. https://doi.org/10.1109/FCCM.2018.00025

[6] Nadesh Ramanathan, George A. Constantinides, and John Wickerson. 2021. Global Analysis of C Concurrency in High-Level Synthesis. *IEEE Trans. Very Large Scale Integr. Syst.* 29, 1 (2021), 24–37. https://doi.org/10.1109/TVLSI.2020.3026112

[7] Nadesh Ramanathan, Shane T. Fleming, John Wickerson, and George A. Constantinides. 2017. Hardware Synthesis of Weakly Consistent C Concurrency. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, Jonathan W. Greene and Jason Helge Anderson (Eds.). ACM, 169–178. http://dl.acm.org/citation.cfm?id=3021733

[8] Nadesh Ramanathan, John Wickerson, and George A. Constantinides. 2018. Scheduling Weakly Consistent C Concurrency for Reconfigurable Hardware. *IEEE Trans. Computers* 67, 7 (2018), 992–1006. https://doi.org/10.1109/TC.2017.2786249

[9] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 209–220. https://doi.org/10.1145/2676726.2676995