Implementing high performance Synchronous Message Exchange University of Copenhagen

Truls Asheim <truls@asheim.dk>

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla laoreet lobortis erat, ac fringilla quam accumsan ut. Nam eget tortor neque. Nulla dictum dapibus venenatis. Vestibulum interdum sem vestibulum enim vulputate ullamcorper nec ut velit. Suspendisse rhoncus a nisi eget viverra. Curabitur porttitor hendrerit elit vel porta. Phasellus elementum nulla aliquet, rhoncus sapien sit amet, maximus libero. Morbi ipsum sapien, faucibus et tempor sed, commodo at enim. Curabitur maximus pretium felis, ac elementum nisi tincidunt et. Sed facilisis lacus purus, sit amet blandit elit ultricies sit amet. Maecenas et cursus libero, ac tincidunt arcu. In at tincidunt ante. Pellentesque lacinia dui nec nulla accumsan auctor faucibus elit suscipit. Pellentesque quis justo turpiso.

Praesent in dignissim lacus, in auctor felis. Phasellus iaculis finibus sapien, vitae vestibulum leo rutrum ac. Etiam in tellus eget sapien suscipit ornare eget sit amet lectus. Vestibulum sagittis consectetur varius. Aenean consectetur ante quis dui iaculis fringilla. Etiam enim sem, facilisis et interdum sit amet, congue nec metus. Fusce eu imperdiet enim, non sollicitudin sem. Vestibulum malesuada mattis justo, vel dictum massa tristique ac. Vivamus pretium turpis ut ligula porta, eget facilisis eros pulvinar. Aenean elit mi, semper vitae ornare vel, pretium sit amet ex. Cras commodo justo quis dolor gravida auctor. Vivamus ultrices arcu dolor, vitae porta ante viverra et. Quisque at aliquet massa, vel gravida felis. Donec eu imperdiet mi. Aliquam in tortor a libero iaculis luctus.

Contents

List of Figures

Chapter 1

Introduction

1.1 Project description

In this report, we describe the design and implementation of a highly efficient library for parallel execution of new, globally synchronous, message passing framework called Synchronous Message Exchange (SME). We will

In this report, we describe a synchronous message passing framework intended to aid simulation of applications whose operating semantics are compatible with the SME model

1.2 Hardware Description Languages

A Hardware Description Language (HDL) is a programming language for describing hardware designs. A program written in such a language is usually

1.3 Background and Motivation

Field-programmable Gate Arrays (FPGA) provides several advantages over using GPGPUS for processing work including a significantly improved performance-per-watt ratio. Due to the low-level nature of current tools for programming FPGAs, their use are largely restricted to engineers with working knowledge in the field of hardware design. In order for software developers to take advantage of FPGAs, improved high-level hardware design utilities are required [1].

The creation of SME was motivated by attempts to use CSP for modeling hardware which, for simple cases, proved successful [3], but additional testing revealed that the CSP model introduced a significant overhead when simulating more complicated hardware designs. Modeling the clock-cycle driven global synchrony that exists in hardware proved to be particularly difficult with CSP and required the introduction of several additional processes and channels. This added complexity reduced simulation performance and limited the usefulness of the CSP-based hardware design model [5].

SME is an attempt to provide a programming framework that, while leveraging and maintaining the properties of CSP that proved useful and f enforcing a hardware-like paradigm, that is accessible to software developers. By [5]

1.4 Limitations

This report will not discuss details related to design of hardware

1.5 Related work

Discuss master thesis

Chapter 2

Analysis and Design

2.1 Synchronous Message Exchange

The system that we will discuss in this report is called Synchronous Message Exchange

2.1.1 Components

Compared to CSP, a much smaller and simpler set of components are used to model the process network. In this section, we describe those components and their

Process A process is an execution unit performing a unit of work

Bus A bus enables communication between two processes and should be considered analogous to buses found in actual hardware. As such, they can only be written to or read from once per clock cycle.

2.1.2 Properties

The SME model has a number of special properties which influences how to efficiently represent and execute the network in our implementation.

Property 1 (Implicit clock)**.** One defining feature of hardware is that all processing is driven by a clock beat. In order to enable fulfillment of this goal we introduce a simulated clock beat in our implementation of SME and thus the defining property of hardware is preserved in the SME model.

Property 2 (Global synchrony)**.** As a consequence of implementing the simulated clock best, all events and communications of the network occurs completely synchronous from the point of view of a process. FIXME

Property 3 (Shared Nothing)**.** A process is completely autonomous and can only change state through receiving a message on its incoming bus. A process is also self contained in the sense

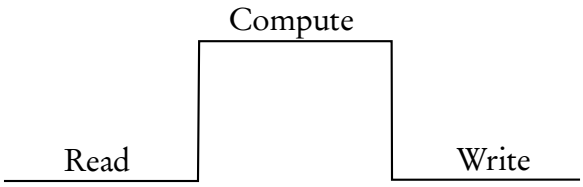

Figure 2.1: The execution cycle of a SME process visualized as a hardware clock-cycle. Before every cycle data from a input bus is read into the process and after a cycle, data is written back to the gate

2.1.3 Execution flow

The execution flow needs to be defined with a focus om preserving the actual
The execution cycle of a SME-process

2.2 Public API

The API exposed to the user of the SME library should be designed with a focus on balancing expressiveness and ability to be easily understandable....

The API implemented by [5] is heavily dependent of the highly dynamic nature of the Python programming language used to implement their prototype. Since our implementation is written in C++ we cannot directly mimic the python API in our implementation. Given the similarities between SME and CSP it seems obvious to let us inspire by

2.3 Paralellization model

Using OS-level threads for representing the SME processes was quickly ruled out during the design process. OS-level threads are severely limited in that the number of threads a process can have is highly limited and switching between OS-level threads is very expensive compared to, for instance, user-level threads. [4].

FiXme Note: Find reference for number of threads per process

Using user-level threads, on linux implemented by using the (now deprecated) `setjump, longjump` functions or the more current `setcontext, getcontext` functions, is another possible way of implementing our networks. The notion of a user-level thread is highly compatible with our concept of a process, namely . User-level threads is the method that is used to implement many CSP libraries, including C++CSP2.

FiXme Note: Find more recent citation for the performance of OS vs. user-level threads

FiXme Note: finish

One important difference between the CSP and SME execution models is that CSP is asynchronous and event-driven in nature while SME is, as previously mentioned, entirely synchronous. This allows us to take a significantly more simple approach to scheduling threads compared to C++CSP. A parallel CSP library needs to schedule a process for execution based on events generated by other processes [2]. In SME we know that every process needs to run during every “clock cycle”. This

FiXme Note: This needs to be better described

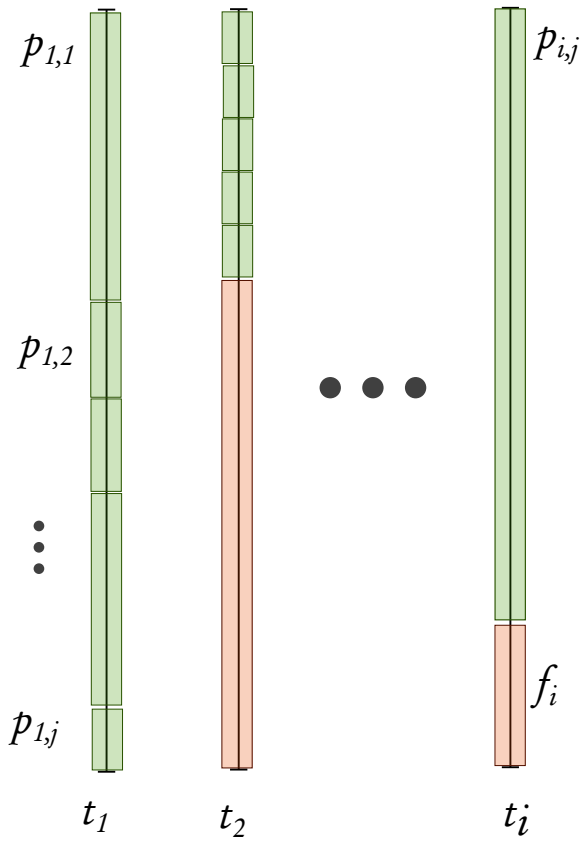

Figure 2.2: Example of suboptimal distribution of processes across processing threads. Green blocks represents processes while red blocks represents thread idle time. Threads are named $p_{i,j}$ where i is the number of the thread the process has been assigned to and f_i is the combined idle time for each thread. Threads are named t_i.

greatly simplifies our implementation of multi-threaded SME.

Based on this, we propose an extremely and efficient model for parallel execution of a SME process network illustrated in. . The idea is to start one OS-level thread per available CPU-core and assign a subset of SME processes to each code. Then, for each iteration of the SME-network , the controlling thread will start by

FiXme Note: make figure similar to Figure 2.2

FiXme Note: Define "iteration of SME-network" as a full clock-cycle of the simulated hardware or replace with different term

2.4 Identifying optimal process scheduling

In order to determine the efficiency of various methods of process scheduling we need to identify the optimality condition for our process scheduling. An illustration of our threading model can be seen in figure 2.2. The green boxes represents processes while the red boxes indicates core idle time. Notice, how we by redistributing the processes across threads could reduce the idle-time of our cores.

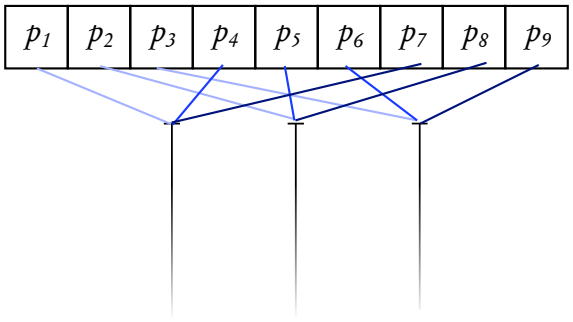

Figure 2.3: Illustration of round-robin process orchestration. Progressive iterations are shown as increasing color intensity of arrows

2.5 Process orchestration

As we discussed in the previous section, the primary limiting factor for our multi-threaded network is an uneven and suboptimal distribution of processes across CPU-cores. If no attempt is made to optimize process distribution, the order of process execution will depend on the order of which processes are defined in the source code. Due to property 3 and property 2 of SME networks there is no scenario where it would be necessary or beneficial for a programmer to exercise ultimate control over the order of process execution. Therefore, maximizing CPU-core utilization would be an unreasonable burden to put on the programmer, especially since their optimization efforts would be specific to a certain number of CPU-cores.

However, the process of orchestrating processes comes at a computational cost which must weighted against the potential reduction of core idle-time.

FiXme Note: is predictability a better word?

The optimal method and timing of process orchestration depends on the dynamicity of the work performed by the network we are executing. A network where each process performs a fixed amount of work per iteration will only need to be orchestrated once, while a network where the workload of the processes are variable will need to be continuously evaluated at runtime in order to maintain our optimality condition. These various methods will be discussed for the remainder of this section.

FiXme Note: Discuss/investigate problem of "optimization-looping". Imagine a network where the execution time of each process is completely random. This would most likely trigger a reorchestration after every iteration which may end up taking more CPU-time than the actual process execution.

2.5.1 Round-robin orchestration

Processes will be executed on the first available core as seen in figure 2.3

2.5.2 One-shot process orchestration

In this model, we orchestrate the processes in our network as soon as possible after execution start and

2.5.3 Monte Carlo orchestration

In this approach, we simply randomize the order of the processes. The main advantage of this approach is that is computationally cheap compared to

2.5.4 Optimization-based orchestration

Another way to orchestrate the processes is to use a

2.5.5 Adaptive process orchestration

The benefits of using a oneshot orchestration approach diminishes when we execute process networks where the processes performs a variable amount of work per iteration. In these kinds of networks, CPU-core load distribution will gradually become uneven and suboptimal as the network execution progresses. In order to keep this from happening and maximize CPU-core utilization, we need to monitor process execution time and core idle time as the network execution progresses. This is what we refer to s adaptive orchestration. This approach, however introduces another trade-off that we need to consider. producing an

2.5.6 Adaptive Monte Carlo process orchestration

2.5.7 Adaptive Optimization-based process orchestration

Chapter 3

Implementation

Chapter 4

Discussion

Bibliography

[1] David F Bacon, Rodric Rabbah, and Sunil Shukla. "FPGA Programming for the Masses". In: *Communications of the ACM* 56.4 (2013), pp. 56–63 (cit. on p. 5).

[2] Neil CC Brown and Peter H Welch. "An introduction to the Kent C++ CSP Library". In: *Communicating Process Architectures 2003* 61 (2003), pp. 139–156 (cit. on p. 8).

[3] Martin REHR, Kenneth SKOVHEDE, and Brian VINTER. "BPU Simulator". In: *Sommunicateind Process Architectures 2013* (2013) (cit. on p. 5).

[4] Minyoung Sung et al. "Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread". In: *Information processing letters* 84.4 (2002), pp. 221–225 (cit. on p. 8).

[5] Brian Vinter and Kenneth Skovhede. "Synchronous Message Exchange for Hardware Designs". In: *Communicating Process Architectures 2014* (2014) (cit. on pp. 5, 6, 8).

Appendix A

Compiling and executing

This section will contain information about how to compile and execute cppsme