Channel communication and Reconfigurable Hardware

Martinus Bos, Paul J.M. Havinga, Gerard J.M. Smit
University of Twente
Department of Computer Science
PO Box 217, 7500 AE Enschede, the Netherlands
phone: +31 53 4893730; fax: +31 53 4894590
E-mail: m.bos@cs.utwente.nl

Abstract—**Many applications can be structured as a set of processes or threads that communicate via channels. These threads can be executed on various platforms (e.g. general purpose CPU, DSP, FPGA, etc). In our research we apply channels as a basic communication mechanism between threads in a reconfigurable system.**

The research involves providing system level functions to describe the setup of communicating threads, which may now either run timeshared on a general CPU or in dynamically-setup special purpose logic that runs on reconfigurable hardware. The use of channels and threads running in both software and hardware, will be made transparent for the application level programmers by the system level functions.

By first describing the threads and how they are connected and then letting the operating system decide on 'geographical' placement of the threads and buffers, multiprogramming will be supported and programs will be able to run on different setups of hardware (i.e. different amount of CPUs or available programmable logic).

This is an ongoing work, the paper is a collection of thoughts, which lead to a first setup of rudimentary support functions in the operating system.

Keywords— **Reconfigurable, csp, operating system, Inferno.**

I. Introduction

Rconfigurable computing. The term hasn't been around for very long yet. It all started with Field Programmable Gate Arrays (FPGAs) that grew so big that at one point they became usable for things other than just glue logic. Things have gone fast since. Many specialized platforms have been built, most using a setup with both FPGAs and a general purpose CPU, but each using its own specific ways to talk to the host system, to carry out reconfigurations and even to store configurations. This is much like coding an application in assembly, or like running a system without Operating System (OS) and rewriting drivers every time. Maybe the OS can make things more comfortable for reconfigurable computing too.

The research involves providing consistent methods for storing configurations (maybe just the normal filesystem to begin with), for distinguishing between different implementations of the same algorithm and for reasoning about these different implementations for Quality of Service purposes. Also, the research is about how to share configurable resources between concurrent applications, how to multi-program them and how to use them for energy efficient computation.

The first basic approach is to provide system level functions to describe the setup of communicating threads, which may now either run time shared on a general CPU or in dynamically-setup special purpose logic that runs on reconfigurable hardware. The use of channels and threads running in both software and hardware, will be made transparent for the application level programmers by the system level functions.

By first describing the threads and how they are connected and then letting the operating system decide on 'geographical' placement of the threads and buffers, multiprogramming will be supported and programs will be able to run on different setups of hardware (i.e. different number of CPUs or different amount of available programmable logic).

II. Software world

Many applications can be structured as a set of processes or threads that communicate via channels. Hoare already used Communicating Sequential Processes (CSP) in [1].

Communicating threads are implemented very straightforward in Lucent's Inferno [2]. They are actually integrated into the Limbo language which is the language to write programs in for Inferno. Limbo includes constructs for both typed channels and threads. A thread is created by the *spawn* keyword, which works much like a normal function call. If a thread is to communicate with other threads, channels can be used and these can be passed as arguments in the spawn call.

The channel mechanism provides both buffered and unbuffered unidirectional communication from one thread to another. It also provides a means of synchronization as channels are blocking such that a writing thread blocks until there is room in the channel buffer and a reading thread blocks until data is available.

On Inferno, threads all share a single CPU. Threads take turn in executing on the CPU. One thread runs on the CPU for a maximum fixed number of instructions or until it executes a blocking system call. Counting instructions is easy as all programs in Inferno run on a virtual machine. When programs are precompiled task switching is no longer forced by counting the number of instructions and thus all threads must give up the CPU voluntarily. The only exception is that systems calls implicitly give up the CPU.

With the mechanism of communicating only through channels, no other complicated mutual exclusion on data structures, critical sections or other synchronization primitives are needed as a thread can be sure it is the only thread running as long as it does not give up the CPU.

In a truly parallel system, one with multiple CPUs or with parallel threads in reconfigurable hardware, this assumption is no longer true. Fortunately, if channels are only used in a pass-by-value manor, i.e. not passing pointers to data, nothing really changes. Parallel systems benefit considerably from many local small memories since these highly increase internal memory bandwidth. So pass-by-value is probably the natural fit in parallel systems as passing pointers is only useful in shared memory systems. In multi-CPU systems the shared memory bandwidth bottleneck can usually be made smaller by the use of on-chip CPU caches. These caches work most efficient when succesive computation on data is executed on the same CPU. Pointer-passing is possible completely transparent when tasks are moved between CPUs, but their gain of moving less data is then lost.

III. Configurable world

In general purpose machines, task-switches occur constantly, 10 millisecond time-slices are normal. With reconfigurable hardware it is likely that task-switches are very expensive. Whereas saving CPU state consists mostly of backing up register contents, saving the state of a reconfigurable piece of hardware includes saving all configuration bits. When memories that can hold multiple configurations which can be swapped instantly are used, one configuration store can be updated while the hardware is running from another. [3] describes prefetching of configuration data on single-context devices and in [4] an architecture is described where multiple configurations are cached. So at least some of the configuration overhead can then be overlapped with useful computation. Reconfiguration can be almost free when runtime is generally much longer than the time needed for storing a configuration. [5] actually shows a technique of compressing configuration data to reduce configuration time.

For this to work in practise, tasks need to be sufficiently big and computations should be regular such that the sequence of configurations is data-independent. When computation is conditional, i.e. different configurations are needed depending on the outcome of current computations, overlapping computation and configuration is no longer possible (unless multiple shadow configutations can be loaded, but that increases configuration time and some configurations would be loaded but never used).

Signal processing algorithms like for example GSM speech-transcoding or video encoding/decoding, in general can be made data-independent when saturating operators are available. With algorithms that perform multiple transformations on blocks of data, a setup where intermediate data is preserved between configurations (one configuration executes one transformation) is possible. This works best when data amounts are big compared to configuration data and the algorithms are latency tolerable. Multiple such local memories connected to the reconfigurable hardware highly increase aggregate memory bandwidth. They do complicate multiprogramming the reconfigurable hardware as saving all state might be close to impossible unless special precautions are taken. A scheduling granularity where tasks are only swapped when internal temporary data is no longer needed seems worth investigating. In [6] an architecture is proposed that interconnects many small memories and reconfigurable tiles in a hierarchical way. In this architecture memories are not really local, but reading/writing memories is more efficient the closer the memories are. Obvious advantage is that the location a certain configuration has to be loaded is not fixed by the location of temporal data from previous computations.

As one algorithm runs best on a DSP and another may run best on an FPGA or CPU, the OS should support a heterogenous mixture of all. In our research we apply channels as a basic communication mechanism between processes in a reconfigurable system. These channels act as the glue between all resources. Occam is one language that includes channels. More recently, they appeared in the language Limbo, that comes with Inferno and just now they have also made their entrance into reconfigurable systems [7]. In Inferno, channels and threads are fully integrated into the system and their use is really natural.

A good example of an application that can make efficient use of channels and threads is a firewall. In a firewall a thread performs some function on the packets coming through. The way these threads are connected is really a streaming setup as packets eventually flow from an input port of the switch to an output port. As firewalls needs constant changing of the functions they perform, and the amounts of data they work on are huge, reconfigurable hardware has the potential to really speed things up.

Taking these huge amounts of state into account it is likely that reconfigurable hardware will be claimed by one and only one task for the duration of that task. Multi-configuration memories can be used both under control by the task (self-reconfiguring programs) and under control by the OS. A compiler directed approach to provide reconfiguration within tasks is described in [8]. In our research scheduling by the OS at points where state is minimal will be investigated. To orchestrate a system in which several tasks are to run simultaneously and each task can potentially be run in different configurations (i.e. performs the same function, but faster/slower, more/less energy efficient), the operating system needs some detailed model upon which it can base its allocation and scheduling decisions.

Although not very common, it seems a good idea to separate running an application into two stages. In the first stage the application consists of one thread running on a general CPU. The application starts negotiating with the OS telling it which modules it needs to run and how these modules communicate with each other. In other words, the application describes to the OS its internal model of communicating processes. The second step of the application is to actually perform the task it is meant to do. As the OS now knows beforehand which modules are needed it can allocate and schedule the resources the application will need and make sure that these resources are efficiently geographically positioned such that all communications are as local as possible.

IV. Status

A first library now exists that allows for experiments with the setup described in the previous sections. The library currently runs on plan9's thread library which has been ported to both

Linux and the Windows operating system. Although not first choice, experiments are now run on Windows, as all tools for the available reconfigurable engine are still not available for any other operating system. Windows is not first choice because it does not, while both plan9 and Linux do, provide sources for the kernel. Kernel support incurs a lot less overhead than a user level library does. To allow future enhancements a lot of work is still needed to get the reconfigurable engine supported in one of the other OS's.

V. Conclusions

As the library does allow describing the model of an algorithm in terms of modules and how these modules are connected through channels, basic allocation schemes can now be developed and tested. Allocation schemes need parameters like circuit size, minimum data throughput, run time and others. Methods to provide these have to be developed and can now be tested. Which parameters a partitioner needs to make valid decisions that can guarantee energy efficiency during runtime and also allow for clear QoS is yet to be determined. The number of variables to base a decision on are likely to grow fast. Extensive simulation of different scenarios should lead to some rules of thumb to allow runtime decision making.

References

[1] C.A.R. Hoare, "Communicating sequential processes", in *Proceedings of the Joint IBM University of Newcastle upon Tyne Seminar*, Newcastle upon Tyne, UK, 1978, pp. 145–56.

[2] Sean Dorward, Rob Pike, David Leo Presotto, Dennis Ritchie, Howard Trickey, and Phil Winterbottom, "Inferno", in *Proceedings of the IEEE Compcon 97 Conference*, San Jose, 1997, pp. 241–244.

[3] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao, "Configuration prefetch for single context reconfigurable processors", in *Proceedings of ACM/SIGDA International Symposium on FPGA*, Feb. 1998.

[4] J.R. Hauser and J.Wawrzynek, "Garp: A mips processor with a reconfigurable coprocessor", in *Proceedings of the IEEE Symposium on FCCM*, Apr. 1997.

[5] S. Hauck, Z. Li, and E.J. Schwabe, "Configuration compression for the xilinx xc6200 fpga", in *Proceedings of the IEEE Symposium on FCCM*, Apr. 1998.

[6] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream computations organized for reconfigurable execution (score)", in *Field-Programmable Logic and Applications, The Roadmap to Reconfigurable Computing*, Reiner W. Hartenstein and Herbert Grunbacher, Eds. Aug. 2000, vol. 1896 of *Lecture Notes in Computer Science*, pp. 605–614, Springer-Verlag.

[7] Jones-M; Scharf-L; Scott-J; Twaddle-C; Yaconis-M; Yao-K; Athanas-P; Schott-B, "Implementing an api for distributed adaptive computing systems", in *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Los Alamitos, CA, USA, 1999, pp. 222–230.

[8] X. Tang, M. Aalsma, and R. Jou, "A compiler directed approach to hiding configuration latency in chameleon processors", in *Field-Programmable Logic and Applications, The Roadmap to Reconfigurable Computing*, Reiner W. Hartenstein and Herbert Grunbacher, Eds. Aug. 2000, vol. 1896 of *Lecture Notes in Computer Science*, pp. 29–38, Springer-Verlag.