BOS: A task-graph processing software framework for the Epiphany many-core architecture

Martin Lundqvist
Ericsson AB
Goteborg, Sweden
martin.lundqvist@ericsson.com

Aare Mällo
Ericsson AB
Goteborg, Sweden
aare.mallo@ericsson.com

Peter Brauer
Ericsson AB
Goteborg, Sweden
peter.brauer@ericsson.com

David Engdal
Ericsson AB
Goteborg, Sweden
david.engdal@ericsson.com

ABSTRACT
This paper outlines a software framework called *BOS* intended for dynamic task graph programming and execution on many-core targets, e.g. the Adapteva Epiphany architecture. An overview of the Epiphany hardware and the application in mind, i.e. radio base-station signal processing, is given. We also present some initial ideas on the resource scheduling problem. Our plan is to release BOS as open source, for further evaluation within the many-core community.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – *Software Frameworks*

I.6.3 [**Simulation and Modeling**]: Applications – *execution time and resource analysis*

C.4 [**Performance of Systems**]: Design studies

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems – *Wireless infrastructure, baseband*

General Terms
Management, Performance, Design, Experimentation

Keywords
Simulation, Many-core, Task Graph, DAG, Real Time, Signal Processing, Baseband, Software Framework

1. INTRODUCTION
As the ICT (Information and Communications Technology) market grows, telecom vendors feel strong customer pressure to increase energy efficiency. The annual energy consumption in the mobile network infrastructure per mobile subscriber is estimated to around 50 kWh [1]. Thus, clearly economic incentive exists to work on power efficiency optimization in this area. With simultaneous requirements on increased data bandwidth telecom vendors are also forced into extensive use of many-core processors [2]. The number of computer cores in a single Ericsson RBS (Radio Base Station) is today typically several hundreds.

Implementing radio base station signal processing on many-core computers has thus been an important part of the R&D work at Ericsson for several years. One good way to formulate this application is as a dynamic task graph processing system. We do this for our current architectures, and other possibly suitable hardware targets should be programmable in the same way.

Being efficient, simple and very scalable, the Epiphany fits our purpose of exploring base band application implementation on promising many-core architectures well. The cores are easily programmed in C/C++ using floating point types, so implementation effort for each compute kernel can be kept to a minimum. Thus we can keep our focus on what is most important: the partitioning and deployment of the application onto the multitude of compute elements. If the Epiphany fits our application and can be used without the occurrence of severe bottlenecks problems, it promises very good performance per energy unit. This can be valid even when compared to regular DSPs which may be a more common choice of target hardware for baseband signal processing applications.

2. BASEBAND APPLICATION
The application used is a part of the baseband processing in wireless communications. The air interface is specified by the 3GPP group, and a large part of the baseband processing is regulated by the standardization. On the other hand, some parts are not specified at all, and the designer may choose the algorithms and calculations.

In this paper, we use part of the LTE uplink processing as our choice for benchmark code. The signal processing code is based on the open source code in [5]. In general, the processing need in an uplink receiver is infinite. It is always possible to get closer to the Shannon limit if more processing power is available.

The processing can be viewed as a graph, with data flowing between nodes, and with barriers to synchronize data handling. See Figure 1 as one example.. These graphs are simple cases of what is known as DAGs (Directed Acyclic Graph) [6].

In the LTE standard, a new task graph set is applied every millisecond (subframe). The air interface scheduler in the RBS decides which mobile units to send data, and how much data in each millisecond. The implication of this is that the processing need for user plane data changes dynamically on millisecond basis.

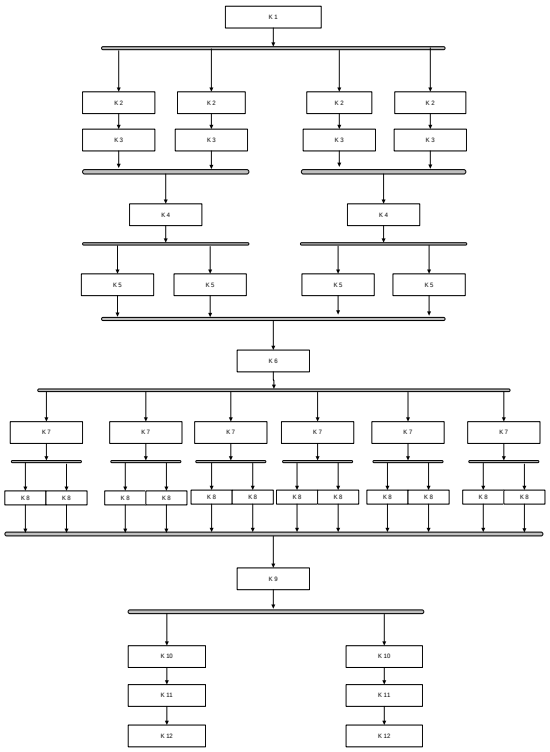

Figure 1 – Example of LTE uplink dataplane flow, one UE, two antennas, two layers.

Each mobile unit (User Equipment, UE) is handled individually in the receiver, resulting in a number of independent, simultaneous graphs competing for the hardware resources. The number of concurrent graphs corresponds to the number of UE's sending in the subframe. The graphs can be very different in data and processing needs.

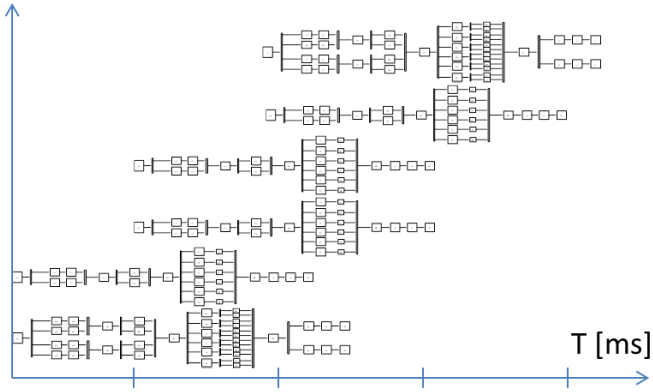

Figure 2 - Example of simultaneous processing of data flows

The LTE standard also specifies when an ACK/NACK must be sent to the UE for received data. The time between receiving and sending defines a latency limit for the data processing. Measurements on a complete RBS determine the latency limits to approximately 2.5 milliseconds for the LTE uplink user plane processing. Since configurations change every millisecond, data from up to three different time-slots may have to be processed simultaneously. Figure 2 is an example with two UE's each subframe.

3. ADAPTEVA – THE EPIPHANY ARCHITECTURE

This section gives a brief introduction to the Epiphany many-core architecture.

3.1 Epiphany: A 2D mesh of simple network and compute elements

The Epiphany is a 2D-mesh many-core architecture with distributed shared memory. The current version of the architecture supports up to 4096 tiles. Each tile consists of a mesh node and a network router. The mesh node, in turn, consist of a RISC CPU, a DMA engine, a local memory (scratch pad, no cache), and a network interface.

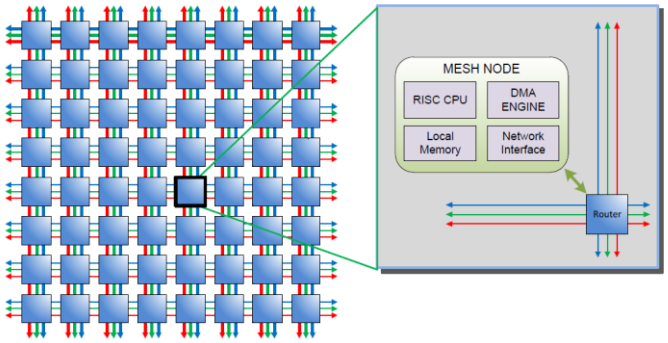

Figure 3 - An example of an implementation of the Epiphany architecture (courtesy of Adapteva) [3].

The RISC CPU ISA is optimized for real-time signal processing and contains 35 instructions. The CPU can execute two floating point operations (i.e. one floating point MAC) and a 64-bit memory load operation on every clock cycle. The local memory consists of 4 banks of 8 kB each, and is addressed by its unique 32-bit address from anywhere on the chip, directly from any of the CPUs or via a DMA engine. The size of the local memory is obviously a limiting factor for both the resident code size of each CPU, and the size of the data blocks handled by a program.

The router is the building block of the Epiphany Network-on-Chip (eMesh). Three separate networks are handled by the router, one for on-chip write traffic, one for off-chip write traffic, and one for all read traffic. The "on-chip write" (called cMesh) is the fastest of these three networks and has a maximum bidirectional throughput of 8 bytes/cycle in each of the four routing directions with zero start up time.

In the latest Epiphany implementation, E64G401, each tile consumes less than 32 mW @ 800 MHz in 28 nm silicon [4].

3.2 Programming the Epiphany

The eCore is fully C/C++-programmable, and given the small instruction set, no one should need to use any intrinsics as is so common and often cumbersome when programming VLIW or SIMD DSPs. No special programming-model is enforced by the architecture; this is to decide for each user/domain. Adapteva supplies an OpenCL compiler which may be very attractive given a suitable problem domain. No task-graph execution platform is yet available.

For further details of the Epiphany, the reader is referred to [3].

4. PROGRAMMING MODEL

When designing the uplink parts of an LTE baseband application for deployment on a many-core platform, the choice of programming model becomes vital. In order to handle the dynamics imposed by the constant changes in configuration, a generalized abstraction of an application graph is introduced.

4.1 Graph anatomy

The first graph in Figure 4 illustrates a basic example of our visualization of signal processing flow, using a loose syntax inspired by different data flow models, e.g. 'Directed Acyclic Graphs'. The arrows represent passing of data, the circles are tasks operating on its respective data, and the bars denote barriers, synchronizing tasks and data. Every task consists of an application specific signal processing function – a kernel – consuming and producing kernel data. Every barrier also comprises the scheduling of resources to execute the following tasks.

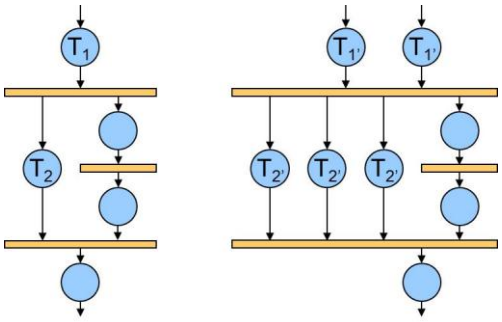

Figure 4 – Latency optimization by parallelization of tasks

4.2 Latency reduction

Depending on the graph configuration, each kernel uses a certain amount of processing to complete its calculations, leading to a certain processing time given the deployment resource properties. In order to fulfill the graph's total latency requirement, an analysis of bottlenecks and parallelization possibilities suggests which kernels to execute in parallel. An example on this can be seen in the second graph in Figure 4, where two tasks were identified as the most beneficial to parallelize.

4.3 Domain separation

Individual tasks have to be independent - the parallelization of any task must not affect any other task. The design of task interfaces will therefore be essential, since they will not only have to offer parallelization possibilities within task boundaries in the graph, but also guarantee task parallelization encapsulation.

These requirements on tasks and interfaces also offer a natural separation of the design domains - signal processing flow and signal processing algorithms. This separation is beneficial, since the different application designers possess different knowledge, have different need for expression, and often use different tools.

4.4 Graph engine – BOS implementation

To enable target execution of graphs on a parallel platform – specifically an Epiphany many-core chip – a basic operating system (BOS) was designed. Certain considerations were taken into account when designing BOS, such as:

- Limited code size – small BOS footprint
- Tight latency requirements – efficient BOS logic
- Visibility – BOS real-time tracing
- Scalability – distributed BOS on each core
- Portability – platform independent BOS

Figure 5 shows the memory structure of one generic core on the Epiphany chip. Due to the nature of our application, we decided to reserve 24 kB of local data for kernel data, while the remaining 8 kB is shared between BOS, kernel code, and a trace buffer – and also some standard basic area definitions.

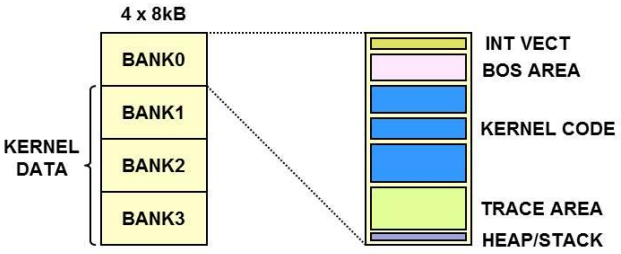

Figure 5 - Memory structure of generic core

The limited memory available makes it impossible for each core to contain code for every application kernel. Instead of e.g. supporting code overlay functionality based on real-time needs, we initially decided to deploy kernel code statically. By predictively analyzing the need for parallelization of kernel functionality and also the passing of data, we tried to optimize the kernel support on the different cores as adequately as possible. Based on execution measurements, we can then hopefully improve our optimization, and perhaps also create rules for how to achieve kernel support rendering it possible for our application to run within all its requirements.

One (or more, if needed) of the cores also supports specific BOS functionality to receive graphs from outside of the Epiphany chip. A received graph is converted from its interface format, sanity checked and stored as a condensed collection of linked lists, as indicated in Figure 6. BOS then starts graph execution by finding a core with kernel support for the first task, moving the required kernel input data to this core's local memory, and finally triggering it to start executing its appointed task.

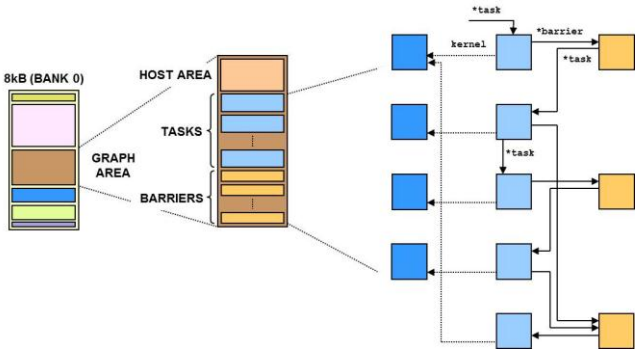

Figure 6 - Graph area storage structure

As soon as a kernel produces output data and finishes, BOS picks up and continues graph execution, according to the basic rules of the programming model, in a totally distributed way. The first task to reach a specific barrier is responsible for finding and reserving cores with kernel support for every task following the barrier. All

tasks reaching a barrier have to copy their kernel output data to these reserved cores, as governed by the graph interfaces, and corresponding to the valid parallelization. When a task in the graph list isn't followed by a barrier, it means that it is the last task in that graph, and that its kernel output data is the graph's final output.

5. RESOURCE ALLOCATION

The resource allocation problem is, somewhat simplified, to decide which part of the code shall run on which core and when.

Based on its configuration, every graph instance obtains an inherent kernel parallelization, of which an example can be seen in the topmost graph of Figure 7. In order to be able to meet the execution requirements an off-line parallelization is performed in a two-step iterative fashion.

First, we parallelize each kernel until code and data physically fits on a single core, and if necessary even further, in order to meet the total latency requirements, resulting in the graph at the bottom left of Figure 7. Then, we continue by adding the constraint of resource limitation, forcing us to stack kernel execution in time, and the final graph resource allocation need can be seen in the bottom right graph.

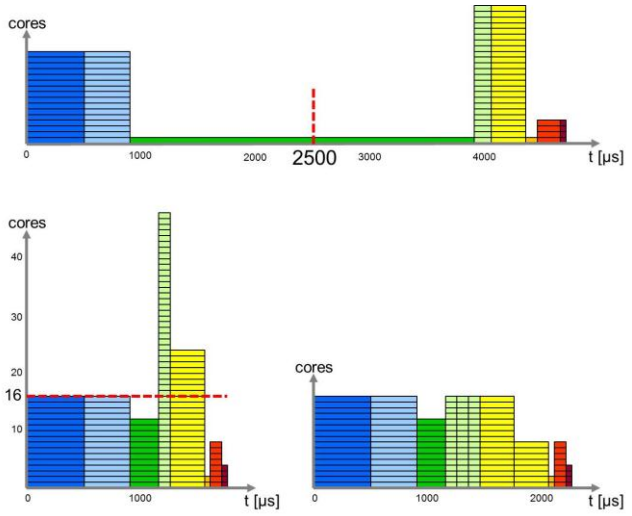

Figure 7 - Task graph parallelization optimization

In run-time we must then decide which core to run a certain instance of a (parallelized) kernel, considering several efficiency criteria. Firstly, no core can support more than a few kernel functions, due to its limited memory size. Subsequently, the moving of data may give rise to data transfer collisions, resulting in long and unpredictable transfer times, depending on how far apart on the chip the interacting cores are situated. The methods to use for this real-time scheduling have not been decided yet, and are subject to ongoing studies.

6. RESULTS

This paper portrays work in progress at Ericsson, and we are currently running BOS on EMEK - an evaluation kit from Adapteva, containing a 16-core version of the Epiphany chip – as well as in a Linux environment, using pthreads for simulating the many-core environment. Within the nearest future – preferably in time for the MCC13 workshop – we will have finished a first round of benchmarking of the Epiphany many-core chip. The results will indicate the capabilities of the Epiphany many-core chip hosting an LTE uplink baseband application, or at least expose its possible weaknesses. We also expect BOS to be scrutinized in the process, and gain experience enough to improve our programming model, and optimize the deployment of it.

7. CONCLUSIONS

As of today, it is too early to draw any confirmed conclusions from what we have accomplished within this benchmarking project. We expect to have completed the first part of our work by the time of the MCC13 workshop, and be able to present valuable experiences in a revised version of this paper as well in a possible workshop presentation.

8. FUTURE WORK

There are several enhancements to BOS possible to implement, however chosen not to pursue within the scope of this first benchmark project.

One improvement would be the on-line ability to optimize a generic graph, given the configuration parameters for a specific graph instance. Thereby we might actually construct every graph using BOS and its resources, thereby minimizing external dependencies.

It is also desirable to improve the basic functionality of BOS, for example the handling of barrier synchronization and moving of kernel data. BOS currently uses a simple search algorithm, flawed but functional, including non-optimal waiting and perhaps inferior dead-lock protection.

Finally, as previously mentioned, a real-time resource scheduler should be added, utilizing run-time resource knowledge.

Our intention is to publish a functioning version of BOS as an open source project as soon as possible. This would make it possible for anyone interested to design their own applications using our programming model, and execute these on Adapteva's Epiphany based products, e.g. their Parallella Board. There is also an opportunity for anyone to feedback comments and suggestions for improvements or even porting to other many-core architectures.

9. REFERENCES

[1] "SMART 2020 : Enabling the Low Carbon Economy in the Information Age," GeSI's Activity Report, The Climate Group on behalf of the Global e-Sustainability Initiative (GeSI), June 2008. Available: http://www.smart2020.org/_assets/files/02_Smart2020Report.pdf

[2] Andras Vajda , (2011), "Programming Many-Core Chips", Springer-Verlag New York Inc., ISBN 9781441997388

[3] EpiphanyTM Architecture Reference, (G3), http://www.adapteva.com/wp-content/uploads/2012/12/epiphany_arch_reference_3.12.12.18.pdf

[4] http://www.adapteva.com/products/silicon-devices/e64g401/

[5] "LTE Uplink Receiver PHY Benchmark," 2011, http://sourceforge.net/projects/lte-benchmark/

[6] Thulasiraman, K.; Swamy, M. N. S. (1992), "5.7 Acyclic Directed Graphs, *Graphs: Theory and Algorithms"*, John Wiley and Son, p. 118, ISBN 978-0-471-51356-8