A Domain-specific Approach for Software Development on Manycore Platforms

Jerker Bengtsson and Bertil Svensson
Centre for Research on Embedded Systems
Halmstad University
PO Box 823, SE-301 18 Halmstad, Sweden
Jerker.Bengtsson@hh.se

Abstract

The programming complexity of increasingly parallel processors calls for new tools that assist programmers in utilising the parallel hardware resources. In this paper we present a set of models that we have developed as part of a tool for mapping dataflow graphs onto manycores. One of the models captures the essentials of manycores identified as suitable for signal processing, and which we use as target for our algorithms. As an intermediate representation we introduce timed configuration graphs, which describe the mapping of a model of an application onto a machine model. Moreover, we show how a timed configuration graph by very simple means can be evaluated using an abstract interpretation to obtain performance feedback. This information can be used by our tool and by the programmer in order to discover improved mappings.

1. Introduction

To be able to handle the rapidly increasing programming complexity of manycore processors, we argue that *domain specific development tools are needed.* The signal processing required in radio base stations (RBS), see figure 1, is naturally highly parallel and described by computations on streams of data [9]. Each module in the figure encapsulates a set of functions, further exposing more pipeline-, data- and task level parallelism as a function of the number of connected users. Many radio channels have to be processed concurrently, each including fast and adaptive coding and decoding of digital signals. Hard real-time constraints imply that parallel hardware, including processors and accelerators is a prerequisite for coping with these tasks in a satisfactory manner.

One candidate technology for building baseband platforms is manycores. However, there are many issues that have to be solved regarding development of complex signal processing software for manycore hardware. One such is

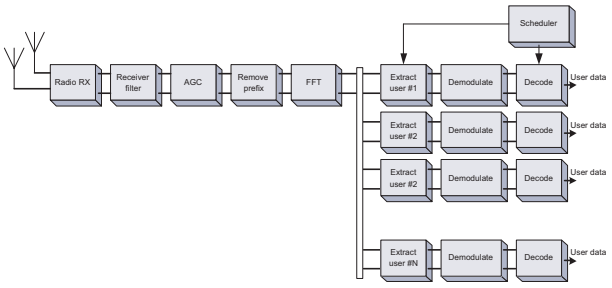

Figure 1. A simplified modular view of the principal functions of the baseband receiver in long term evolution (LTE) RBS.

the need for tools that reduce the programming complexity and abstract the hardware details of a particular manycore processor. We believe that if industry is to adopt manycore technology *the application software, the tools and the programming models need to be portable*.

Research has produced efficient compiler heuristics for programming languages based on streaming models of computation (MoC), achieving good speedup and high throughput for parallel benchmarks [3]. However, even though a compiler can generate optimized code the programmer is left with very little control of how the source program is transformed and mapped on the cores. This means that if the resulting code output does not comply with the system timing requirements, the only choice is to try to restructure the source program. We argue that *experienced application programmers must be able to direct and specialize the parallel mapping strategy by giving directive tool input*.

For complex real-time systems, such as baseband processing platforms, we see a need for tunable code parallelization- and mapping tools, allowing programmers to take the system's real-time properties into account during the optimization process. Therefore, complementary to

fully automatized parallel compilers, we are proposing an iterative code parallelization- and mapping tool flow that allows the programmer to tune mapping by:

- analyzing the result of a parallel code map using performance feedback
- giving timing constraints, clustering and core allocation directives as input to the tool

In our work we address the design and construction of one such tool. We focus on suitable well defined dataflow models of computation for modeling applications and manycore targets, as well as the base for our intermediate representation for manycore code-generation. One such model, synchronous dataflow (SDF), is very suitable for describing signal processing flows. It is also a good source for code-generation, given that it has a natural form of parallelism that is a good match to manycores. The goal of our work is a tool chain that allows the software developer to specify a manycore architecture (using our *machine model*), to describe the application (using SDF) and to obtain a generated mapping that can be evaluated (using our *timed configuration graph*). Such a tool allows the programmer to explore the run time behaviour of the system and to find successively better mappings. We believe that this iterative, machine assisted, workflow, is good in order to keep the application portable while being able to make trade-offs concerning throughput, latency and compliance with real-time constraint on different platforms.

In this paper we present our set of models and show how we can analyze the mapping of an application onto a manycore. More specifically, the contributions of this paper are as follows:

- A parallel machine model usable for modelling array-structured, tightly coupled manycore processors. The model is presented in Section 2, and in Section 3 we demonstrate modeling of one target processor.
- A graph-based intermediate representation (IR), used to describe a mapping of an application on a particular manycore in the form of a (*a timed configuration graph*). The use of this IR is twofold. We can perform an abstract interpretation that gives us feedback about the dynamic behaviour of the system. Also, we can use it to generate target code. We present the IR in Section 4.
- We show in Section 5 how parallel performance can be evaluated through abstract interpretation of the timed configuration graph. As a proof of concept we have implemented our interpreter in the Ptolemy II software framework using dataflow process networks.

We conclude our paper with a discussion of our achievements and future work.

2 Model Set

In this section we present the model set for constructing *timed configuration graph*s. First we discuss the application model, which describes the application processing requirements, and then the machine model, which is used to describe computational resources and performance of manycore targets.

2.1 Application Model

We model an application using SDF, which is a special case of a computation graph [5]. An SDF graph constitutes a network of actors - atomic or composite of variable granularity - which asynchronously compute on data distributed via synchronous uni-directional channels. By definition, actors in an SDF graph fire (compute) in parallel when there are enough tokens available on the input channels. An SDF graph is computable if there exists at least one static repetition schedule. A repetition schedule specifies in which order and how many times each actor is fired. If a repetition schedule exists, buffer boundedness and deadlock free execution is guaranteed. A more detailed description of the properties of SDF and how repetition schedules are calculated can be found in [6].

The Ptolemy II modelling software provides an excellent framework for implementing SDF evaluation- and code generator tools [1]. We can very well consider an application model as an executable specification. For our work, it is not the correctness of the implementation that is in focus. We are interested in analyzing the dynamic, non-functional behaviour of the system. For this we rely on measures like worst case execution time, size of dataflows, memory requirements etc. We assume that these data have been collected for each of the actors in the SDF graph and are given as a tuple

$$< r_p, r_m, R_s, R_r >$$

where

- r_p is the worst case computation time, in number of operations.
- r_m is the requirement on local data allocation, in words.
- $R_s = [r_{s_1}, r_{s_2}, ..., r_{s_n}]$ is a sequence where r_{s_i} is the number of words produced on channel i each firing.
- $R_r = [r_{r_1}, r_{r_2}, ..., r_{r_m}]$ is a sequence r_{r_j} is the number of words consumed on channel j each firing.

2.2 Machine Model

One of the early, reasonably realistic, models for distributed memory multiprocessors, is the LogP model [2]. Work has been done to to refine this model, for example taking into account hardware support for long messaging, and to capture memory hierarchies. A more recent parallel machine model for multicores, which considers different core granularities and requirements on on-chip and off-chip communication is Simplefit [7]. However, this model was derived with the purpose of exploring optimal grain size and balance between memory, processing, communication and global I/O, given a VLSI budget and a set of computation problems. Since it is not intended for modeling dynamic behaviour of a program, it does not include a fine-granular model of the communication. Taylor et al. propose a taxonomy (AsTrO) for comparison of scalar operand networks [11]. They also provide a tuple based model for comparing and evaluating performance sensitivity of on-chip network properties.

We propose a manycore machine model based on Simplefit and the AsTrO taxonomy, which allows a fairly fine-grained modeling of parallel computation performance including the overhead of operations associated with communication. The machine model comprises a set of parameters describing the computational resources and a set of abstract performance functions, which describe the computational performance of computations, communication and memory transactions. We will later show in Section 5 how we can can model dynamic, non-functional behavior of a dataflow graph mapped on a manycore target, by incorporating the machine model in a dataflow process network.

2.2.1 Machine Specification

We assume that cores are connected in a mesh structured network. Further that each core has individual instruction decoding capability and software managed memory load and store functionality, to replace the contents of core local memory. We describe the resources of such a manycore architecture using two tuples, M and F. M consists of a set of parameters describing the processors resources:

$$M =< (x, y), p, b_g, g_w, g_r, o, s_o, s_l, c, h_l, r_l, r_o >$$

where

- (x, y) is the number of rows and columns of cores.
- p is the processing power (instruction throughput) of each core, in *operations per clock cycle*.
- b_g is global memory bandwidth, in *words per clock cycle*
- g_w is the penalty for global memory write, in *words per clock cycle*
- g_r is the penalty for global memory read, in *words per clock cycle*
- o is software overhead for initiation of a network transfer, in *clock cycles*
- s_o is core send occupancy, in *clock cycles*, when sending a message.
- s_l is the latency for a sent message to reach the network, in *clock cycles*
- c is the bandwidth of each interconnection link, in *words per clock cycle*.
- h_l is network hop latency, in *clock cycles*.
- r_l is the latency from network to receiving core, in *clock cycles*.
- r_o is core receive occupancy, in *clock cycles*, when receiving a message

F is a set of abstract functions describing the performance of computations, global memory transactions and local communication:

$$F(M) =< t_p, t_s, t_r, t_c, t_{gw}, t_{gr} >$$

where

- t_p is a function evaluating the time to compute a list of instructions
- t_s is a function evaluating the core occupancy when sending a data stream
- t_r is a function evaluating the core occupancy when receiving a data stream
- t_c is a function evaluating network propagation delay for a data stream
- t_{gw} is a function evaluating the time for writing a stream to global memory
- t_{gr} is a function evaluating the time for reading a stream from global memory

A specifc manycore processor is modeled by giving values to the parameters of M and by defining the functions $F(M)$.

3 Modeling the RAW Processor

In this section we demonstrate how we configure our machine model in order to model the RAW processor [10]. RAW is a tiled, moderately parallel MIMD architecture with 16 programmable tiles, which are tightly connected via two different types of communication networks: two statically- and two dynamically routed. Each tile has a MIPS-type pipeline and is equipped with 32 KB of data and 96 KB instruction caches.

3.1 Parameter Settings

We are assuming a RAW setup with non-coherent off-chip global memory (four concurrently accessible DRAM banks), and that software managed cache mode is used. Furthermore, we concentrate on modeling usage of the dynamic networks, which are dimension-ordered, wormhole-routed, message-passing type of networks. The parameters of M for RAW with this configuration are as follows:

$$\begin{aligned} M =< \quad (x,y) &= (4,4), \\ p &= 1, \\ b_g &= 1, \\ g_w &= 1, \\ g_r &= 6, \\ o &= 2, \\ s_o &= 1, \\ s_l &= 1, \\ c &= 1, \\ h_l &= 1, \\ r_l &= 1, \\ r_o &= 1 > \end{aligned}$$

In our model, we assume a core instruction throughput of p operations per clock cycle. Each RAW tile has an eight-stage, single-issue, in-order RISC pipeline. Thus, we set $p = 1$. An uncertainty here is that in our current analyses, we cannot account for pipeline stalls due to dependencies between instructions having non-equal instruction latencies. We need to make further practical experiments, but we believe that this in general will be averaged out equally on cores and thereby not have too large effects on the estimated parallel performance.

There are four shared off-chip DRAMs connected to the four east-side I/O ports on the chip. The DRAMs can be accessed in parallel, each having a bandwidth of $b_g = 1$ words per clock cycle per DRAM. The penalty for a DRAM write is $g_w = 1$ cycle and correspondingly for read operation $g_r = 6$ cycles.

Since the communication patterns for dataflow graphs are known at compile time, message headers can be precomputed when generating the communication code. The overhead includes sending the header and possibly an address (when addressing off-chip memory). We therefore set $o = 2$ for header and address overhead when initiating a message.

The networks on RAW are mapped to the core's register files, meaning that after a header has been sent, the network can be treated as destination or source operand of an instruction. Ideally, this means that the receive and send occupancy is zero. In practice, when multiple input and output dataflow channels are merged and physically mapped on a single network link, data needs to be buffered locally. Therefore we model send and receive occupancy – for each word to be sent or received – by $s_o = 1$ and $r_o = 1$ respectively. The network hop-latency is $h_l = 1$ cycles per hop and the link bandwidth is $c = 1$. Furthermore, the send and receive latency is one clock cycle when injecting and extracting data to and from the network: $s_l = 1$ and $r_l = 1$ respectively.

3.2 Performance Functions

We have derived the performance functions by studying the hardware specification and by making small comparable experiments on RAW. We will now show how the performance functions for RAW are defined.

Compute The time required to process the fire code of an actor on a core is expressed as

$$t_p(r_p, p) = \left\lceil \frac{r_p}{p} \right\rceil$$

which is a function of the requested number of operations r_p and core processing power p. To r_p we count all instructions except those related to network send- and receive operations.

Send The time required for a core to issue a network send operation is expressed as

$$t_s(R_s, o, s_o) = \left\lceil \frac{R_s}{\mathit{framesize}} \right\rceil \times o + R_s \times s_o$$

Send is a function of the requested amount of words to be sent, R_s, the software overhead $o \in M$ when initiating a network transfer, and a possible send occupancy $s_o \in M$. The $framesize$ is a RAW specific parameter. The dynamic networks allow message frames of length within the interval $[0, 31]$ words. For global memory read and write operations, we use RAWs cache line protocol with $framesize = 8$ words per message. Thus, the first term of t_s captures the software overhead for the number of messages required to send the complete stream of data. For connected actors that are mapped on the same core, we can choose to map channels in local memory. In that case we set t_s to o zero time.

Receive The time required for a core to issue a network receive operation is expressed as

$$t_r(R_r, o, r_o) = \left\lceil \frac{R_r}{framesize} \right\rceil \times o + R_r \times r_o$$

The receive overhead is calculated in a similar way as the send overhead, except that parameters of the receiving core replace the parameters of the sending core.

Network Propagation Time Modeling shared resources accurately with respect to contention effects is very difficult. Currently, we assume that SDF graphs are mapped so that the communication will suffer from no or a minimum of contention. In the network propagation time, we consider a possible network injection- and extraction latency at the source and destination as well as the link propagation time. The propagation time is expressed as

$$t_c(R_s, d, s_l, h_l, r_l) = s_l + d \times h_l + n_{turns} + r_l$$

Network injection- and extraction latency is captured by s_l and r_l respectively. Further, the propagation time is dependent on the network hop latency h_l and the number of network hops d, which are determined from the source and destination coordinates as $|x_s - x_d| + |y_s - y_d|$. Routing turns add an extra cost of one clock cycle. This is captured by the value of n_{turns} which, similar to d, is calculated using the source and destination coordinates.

Streamed Global Memory Read Reading from global memory on the RAW machine requires first one send operation (the core overhead which is captured by t_s), in order to configure the DRAM controller and set the address of memory to be read. The second step is to issue a receive operation to receive the memory contents on that address. The propagation time when streaming data from global memory to the receiving core is expressed as

$$t_{gr} = r_l + d \times h_l + n_{turns}$$

Note that memory read penalty is not included in this expression. This is accounted for in the memory model included in the IR. This is further discussed in Section 4

Streamed Global Memory Write Similarly to the memory read operation, writing to global memory require two send operations: one for configuring the DRAM controller (set write mode and address) and one for sending the data to be stored. The time required for streaming data from the sending core to global memory is evaluated by

$$t_{gw} = s_l + d \times h_l + n_{turns}$$

Like in stream memory read, the memory write penalty is accounted for in the memory model.

4 Timed Configuration Graphs

In this section we describe our manycore intermediate representation (IR). We call the IR a *timed configuration graph* because the usage of the IR is twofold:

- Firstly, the IR is a graph representing an SDF application graph, after it has been clustered and partitioned for a specific manycore target. We can use the IR as input to a code generator, in order to configure each core as well as the interconnection network and plan global memory usage of a specific manycore target.
- Secondly, by introducing the notion of time in the graph, we can use the same IR as input to an abstract interpreter, in order to evaluate the dynamic behaviour of the application when executed on a specific manycore target. The output of the evaluator can be used either directly by the programmer or to extract information feedback to the tool for suggesting a better mapping.

4.1 Relations Between Models and Configuration Graphs

A *configuration graph* $G_M^A(V, E)$ describes an application A mapped on the abstract machine M. The set of vertices $V = P \cup B$ consists of cores $p \in P$ and global memory buffers $b \in B$. Edges $e \in E$ represent dataflow channels mapped onto the interconnection network. To obtain a G_M^A, the SDF for A is partitioned into subgraphs and each subgraph is assigned to a core in M. The edges of the SDF that end up in one subgraph are implemented using local memory in the core, so they do not appear as edges in G_M^A. The edges of the SDF that reach between subgraphs can be dealt with in two different ways:

1. A network connection between the two cores is used and this appears as an edge in G_M^A
2. Global memory is used as a buffer. In this case, a vertex b (and associated input- and output edges) is introduced between the two cores in G_M^A.

When G_M^A has been constructed, each $v \in V$ and $e \in E$ has been assigned computation times and communication delays, calculated using the parameters of M and the performance functions $F(M)$ introduced in Section 2.2. These annotations reflect the performance when computing the application A on the machine M. We will now discuss how we use A and M to configure the vertices, edges and then computational delays of G_M^A.

4.1.1 Vertices.

We distinguish between two types of vertices in G_M^A: *memory* vertices and *core* vertices. Introducing *memory* vertices allows us to represent global memory. A *memory* vertex can be specified by the programmer, for example to store initial data. More typically, *memory* vertices are automatically generated when mapping channel buffers in global memory.

For *core* vertices, we abstract the firing of an actor by means of a sequence S of abstract *receive*, *compute* and *send* operations:

$$S = t_{r_1}, t_{r_2} \ldots t_{r_n}, t_p, t_{s_1}, t_{s_2}, \ldots, t_{s_m}$$

The *receive* operation has a delay corresponding to the timing expression t_r, representing the time for an actor to receive data through a channel. The delay of a *compute* operation corresponds to the timing expression t_p, representing the time required to execute the computations of an actor when it fires. Finally, the *send* operation has a delay corresponding to the timing expression t_s, representing the time for an actor to send data through a channel.

For a *memory* type of vertex, we assign delays specified by g_r and g_w in the machine model to account for memory read- and write latencies respectively.

When building G_M^A, multiple channels sharing the same source and destination can be merged and represented by a single edge, treating them as a single block or stream of data. Thus, there is always only one edge $e_{i,j}$ connecting the pair (v_i, v_j). We add one *receive* operation and one *send* operation to the sequence S for each input and output edge respectively.

4.1.2 Edges.

Edges represent dataflow channels mapped onto the interconnection network. The weight w of an edge $e_{i,j}$ corresponds to the communication delay between vertex v_i and vertex v_j. The weight w depends on whether we map the channel as a point-to-point data stream over the network, or in shared memory using a *memory* vertex.

In the first case we assign the edge a weight corresponding to t_c. When a channel buffer is placed in global memory, we substitute the edge in A by a pair of input- and output edges connected to a *memory* actor. We illustrate this by Figure 2. We assign a delay of t_{gr} and t_{gw} to the input and output edges of the *memory* vertex.

Figure 3 shows an example of a simple A transformed to one possible G_M^A. A repetition schedule for A in this example is $3(2ABCD)E$. The repetition schedule should be interpreted as: actor A fires 6 times, actors B, C and D fire 3 times, and actor E 1 time. The firing of A is repeated indefinitly by this schedule. We use dashed lines for actors of A mapped and translated to S inside each core vertex of G_M^A. The feedback channel from C to B is mapped

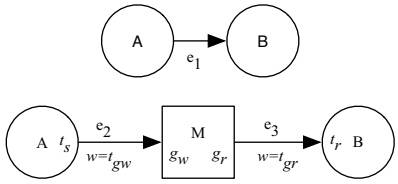

Figure 2. The lower graph (G_M^A) in the figure illustrates how the unmapped channel e_1, connecting actor A and actor B, in the upper graph (A), has been transformed and replaced by a global memory actor and edges e_2 and e_3.

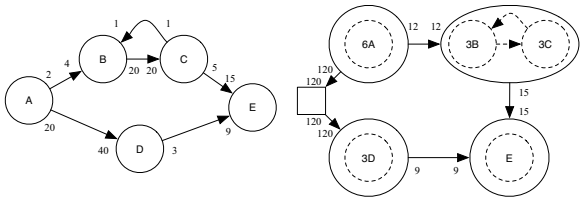

Figure 3. The graph to the right is one possible G_M^A for the graph A to the left.

in local memory. The edge from A to D is mapped via a global buffer and the others are mapped as point-to-point data streams. The integer values represent the send and receive rates of the channels (r_s and r_r), before and after A has been clustered and transformed to G_M^A, respectively. Note that these values in G_M^A are the values in A multiplied by the number of the repetition schedule.

5 Interpretation of Timed Configuration Graphs

In this section we show how we can make an abstract interpretation of the IR and how an interpreter can be implemented by very simple means on top of a dataflow process network. We have implemented such an interpreter using the dataflow process networks (PN) domain in Ptolemy. The PN domain in Ptolemy is a super set of the SDF domain. The main difference in PN, compared to SDF, is that PN processes fire asynchronously. If a process tries to read from an empty channel, it will block until there is new data available. The PN domain implemented in Ptolemy is a special case of Kahn process networks [4]. Unlike in a Kahn process network, PN channels have bounded buffer capacity, which implies that a process also temporarily blocks

when attempting to write to a buffer that is full [8]. This property makes it possible to easily model link occupancy on the network. Conclusively, a dataflow process network model perfectly mimics the behavior of the types of parallel hardware we are studying. Thus, a PN model is a highly suitable base for an intermediate abstraction for the processor we are targetting.

5.1 Parallel Interpretation using Process Networks

Each of the core and memory vertices of G_M^A is assigned to its own process. Each of the core and memory processes has a local clock, t, which iteratively maps the absolute start and stop time, as well as periods of blocking, to each operation in the sequence S.

A core process evaluates a vertex by means of a state machine. In each clock step, the current *state* is evaluated and then stored in the *history*. The *history* is a chronologically ordered list describing the *state* evolution from time $t = 0$.

5.2 Local Clocks

The clock t is process local and stepped by means of (not equal) time segments. The length of a time segment corresponds to the delay bound to a certain operation or the blocking time of a send or receive operation. The execution of send and receive operations in S is dependent on when data is available for reading or when a channel is free for writing, respectively.

5.3 States

For each vertex, we record during what segments of time computations and communication operations were issued, as well as periods where a core has been stalled due to send- and receive blocking. For each process, a *history* list maps to a state $type \in Stateset$, a start time t_{start} and a stop time t_{stop}. The $state$ of a vertex is a tuple

$$state =< type, t_{start}, t_{stop} >$$

The *StateSet* defines the set of possible state types:

$$StateSet = \{receive, compute, send, \\ blockedreceive, blockedsend\}$$

5.4 Clock Synchronisation

Send and receive are blocking operations. A read operation blocks until data is available on the edge and a write operation blocks until the edge is free for writing. During a time segment only one message can be sent over an edge. Clock synchronisation between communicating processes is managed by means of *event*s. Send and receive operations generate an *event* carrying a time stamp. An edge in G_M^A is implemented using channels having buffer size 1 (forcing write attempts on an occupied link to block), and a simple delay actor. It should be noted that each edge in A needs to be represented by a pair of opposite directed edges in G_M^A to manage synchronization.

receive($t_{receive}$)
  $t_{available}$ = get next send event from source vertex
  if($t_{receive} >= t_{available}$)
    $t_{read} = t_{receive+1}$
    $t_{blocked} = 0$
  else
    $t_{read} = t_{available+1}$
    $t_{blocked} = t_{available} - t_{receive}$
  end if
  put read event with time t_{read} to source vertex
  return $t_{blocked}$
end

Figure 4. Pseudo-code of the receive function. The get and put operations block if the event queue of the edge is empty or full, respectively.

5.4.1 Synchronised Receive

Figure 4 lists pseudo code of the blocking *receive* function. The value of the input $t_{receive}$ is the present time at which a receiving process issues a *receive* operation. The return value, $t_{blocked}$, is the potential blocking time. The time stamp $t_{available}$, is the time at which the message is available at the receiving core. If $t_{receive}$ is later or equal to $t_{available}$, the core immediately processes the receive operation and sets $t_{blocked}$ to 0. The *receive* function acknowledges by sending a read event to the sender, with the time stamp t_{read+1}. Note that a channel is free for writing as soon as the reciver has begun receiving the previous message. Also note that blocking time, due to unbalanced production and consumption rates, has been accounted for when analysing the timing expression for *send* and *receive* operations, T_s and T_r, as was discussed in Section 2.2. If $t_{receive}$ is earlier than $t_{available}$, the receiving core will block a number of clock cycles corresponding to $t_{blocked} = t_{available} - t_{receive}$.

5.4.2 Synchronised Send

Figure 5 lists pseudo code for the blocking *send* function. The value of t_{send} is the time at which the *send* operation was issued. The time stamp of the read event $t_{available}$ corresponds to the time at which the receiving vertex reads the previous message and thereby also when the edge is available for sending next message. If $t_{send} < t_{available}$, a *send* operation will block for $t_{blocked} = t_{available} - t_{send}$ clock cycles. Otherwise $t_{blocked}$ is set to 0. Note that all edges carrying receive events in the *configuration graph* must be initialised with a read event, otherwise interpretation will deadlock.

```
send(t_send)
  t_available = get read event from sink vertex
  if(t_send < t_available)
    t_blocked = t_available − t_send
  else
    t_blocked = 0
  end if
  put send event t_send + Δ_e + t_blocked to sink vertex
  return t_blocked
end
```

Figure 5. Pseudo-code of the send function. The value of Δ_e corresponds to the delay of the edge.

5.5 Vertex Interpretation

Figure 6 lists the pseudo code for interpretation of a vertex in G_M^A. It should be noted that, for space reasons, we have omitted to include the state code for global read and write operations. The function $interpretVertex()$ is finitely iterated by each process and the number of iterations, $iterations$, is equally set for all vertices when processes are initated. Each process has a local clock t and an operation counter op_cnt, both initially set to 0. The operations sequence S is a process local data structure, obtained from the vertex to be interpreted. Furthermore, each process has a list $history$ which initially is empty. Also, each process has a variable $curr\_oper$ which holds the currently processed operation in S.

The vertex interpreter makes state transitions depending on the current operation $curr\_oper$, the associated delay and whether *send* and *receive* operations block or not. As discussed in Section 5.4.1, the *send* and *receive* functions are the only blocking functions that can halt the interpretation in order to synchronise the clocks of the processes. The value of $t_{blocked}$ is set to the return value of *send* and *receive* when interpreting send and receive operations, respectively. The value of $t_{blocked}$ corresponds to the length of time a *send* or *receive* operation was blocked. If $t_{blocked}$ has a value > 0, a state of type *blocked_send* or *blocked_receive* is computed and added to the *history*.

```
interpretVertex()
  if(list S has elements)
    while(iterations > 0)
      get element op_cnt in S and put in curr_oper
      increment op_cnt

      if(curr_op is a Receive operation)
        set t_blocked = value of receive(t)
        if(t_blocked > 0)
          add state ReceiveBlocked(t, t_blocked) to hist.
          set t = t + t_blocked
        end if
        add state Receiving(t, Δ of curr_oper)
      end if

      else if(curr_op is a Compute operation)
        add state Computing(t, Δ of curr_oper)
      end if

      else if(curr_op is a Send operation)
        set t_blocked = value of send(t)
        if(t_blocked > 0)
          add state SendBlocked(t, t_blocked) to hist.
          set t = t + t_blocked
        end if
        add state Sending(t, Δ of curr_oper)
      end if

      if(op_cnt reached last index of S)
        set op_cnt = 0
        decrement iterations
        add state End(t) to history
      end if
      set t = t + Δ of curr_oper + 1
    end while
  end if
end
```

Figure 6. Pseudo-code of the interpretVertex function.

5.6 Model Calibration

We have implemented the abstract interpreter in the Ptolemy software modeling framework [1]. Currently, we have verified the correctness of the interpreter using a set of simple parallel computation problems from the literature. Regarding the accuracy of the model set, we have so far only compared the performance functions separately against corresponding operations on RAW. However, to evaluate and possibly tune the model for higher accuracy we need to do further experimental tests with different relevant signal processing benchmarks, especially including some more complex communication- and memory access patterns.

6 Discussion

We believe that tools supporting iterative mapping and tuning of parallel programs on manycore processors will play a crucial role in order to maximise application performance for different optimization criteria, as well as to reduce the parallel programming complexity. We also believe that using well defined parallel models of computation, matching the application, is of high importance in this matter.

In this paper we have presented our achievements towards the building of an iterative manycore code generation tool. We have proposed a machine model, which abstracts the hardware details of a specific manycore and provides a fine-grained instrument for evaluation of parallel performance. Furthermore, we have introduced and described an intermediate representation called *timed configuration graph*. Such a graph is annotated with computational delays that reflect the performance when the graph is executed on the manycore target. We have demonstrated how we compute these delays using the performance functions included in the machine model and the computational requirements captured in the application model. Moreover, we have in detail demonstrated how performance of a *timed configuration graph* can be evaluated using abstract interpretation.

As part of future work, we need to perform further benchmarking experiments in order to better determine the accuracy of our machine model compared to chosen target processors. Also, we have so far built *timed configuration graph*s by hand. We are especially interested in exploring tuning methods, using feedback information from the evaluator to set constraints in order to direct and improve the mapping of application graphs. Currently we are working on automatising the generation of the *timed configuration graph*s in our tool-chain, implemented in the Ptolemy II software modelling framework.

Acknowledgment

The authors would like to thank Henrik Sahlin and Peter Brauer at the Baseband Research group at Ericsson AB, Dr. Veronica Gaspes at Halmstad University, and Prof. Edward A. Lee and the Ptolemy group at UC Berkeley for valuable input and suggestions. This work has been funded by research grants from the Knowledge Foundation under the CERES contract.

References

[1] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2008-28, EECS Dept., University of California, Berkeley, Apr 2008.

[2] D. Culler, R. Karp, and D. Patterson. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. of ACM SIGPLAN Symposium on Principles and Practices of Parallel programming*, May 1993.

[3] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in stream programs. In *Proc. of Twelfth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[4] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 5-10 1974. North-Holland Publishing Company.

[5] R. M. Karp and R. E. Miller. Properties of a Model for Parallel Computations:Determinancy, Termination, Queueing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, November 1966.

[6] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Signal Processing. *IEEE Trans. on Computers*, January 1987.

[7] C. A. Moritz, D. Yeung, and A. Agarwal. SimpleFit: A Framework for Analyzing Design Tradeoffs in Raw Architectures. *IEEE Trans. on Parallel and Distributed Systems*, 12(6), June 2001.

[8] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Dept., University of California, Berkeley, Berkeley, CA, USA, 1995.

[9] H. Sahlin. Introduction and overview of LTE Baseband Algorithms. Powerpoint presentation, Baseband research group, Ericsson AB, February 2007.

[10] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[11] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar Operand Networks. *IEEE Trans. on Parallel and Distributed Systems*, 16(2):145–162, 2005.