Technical Report **IDE0856**, July 2008

A Set of Models for Manycore Performance Evaluation through Abstract Interpretation of Timed Configuration Graphs

Jerker Bengtsson and Bertil Svensson

School of Information Science, Computer and Electrical Engineering,
Halmstad University, Box 823, SE-30118 Halmstad, Sweden

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 a 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* 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.

Contents

1 Introduction

Simultaneous processing of many parallel radio channels in next generation high-end radio base stations require fast, adaptive, real-time coding and decoding of digital signals. Different types and levels of parallelism can also be found with the baseband algorithms. To manage processing for a frequently varying set of simultaneous users, with different requirements on quality of service, there is a need for flexible (programmable) platform solutions. The varying workload in combination with the hard real-time constraints, given by the baseband protocol standards, require carefully optimised parallel processing.

Manycores is a good candidate technology for baseband processing platforms. However, there are many issues that have to be solved regarding development of complex signal processing software for manycore hardware. One such is 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 manycores the application software, the tools and the programming models need to be portable .

In our work we address the design and construction of one such tool. We focus on well defined dataflow models of computation and build a manycore code-generator using a number of different models. 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 successive 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 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 abstraction usable for modelling array-structured,

tightly coupled manycore processors. We present it in Section 2.

- A graph-based intermediate representation (IR) used to describe a mapping of an application on a particular manycore (*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 3.

- We show in Section 4 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 our 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 describes 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 [1]. Special, because SDF graphs are not terminating and the sizes of all dataflow buffers are deterministic. An SDF graph constitutes a network of actors - atomic or composite of variable granularity - which computes on data distributed via uni-directional input- and output 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 are fired. If a repetition schedule exists, buffer boundedness and deadlock free execution is guaranteed. A more detailed description of the properties of SDF graphs can be found in [2].

The Ptolemy II modelling software provides an excellent framework for implementing SDF evaluation- and code generator tools [3]. 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 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

$$< r_p, r_m, r_{s_i}, r_{r_j} >$$

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_i} is the number of words produced on channel i each firing.
- r_{r_j} is the number of words consumed on channel j each firing.

2.2 Machine Model

The term *manycore* was coined at Berkeley to distinguish new paradigms of highly parallel processors from mainstream *multicores*, which is the term adopted by the general purpose micro-processor industry [4]. We are interested in processors where the cores and the interconnection network are tightly integrated on a chip, offering a reduced communication overhead [5]. Traditionally hardware implemented functionality, such as caching, typically is removed and must be managed by software. Processors with tightly coupled cores have potential for better exploitation of finer-grained parallelism, due to reduced communication overhead and optimisation of memory usage.

One of the early, reasonably realistic, models for distributed memory multiprocessors, is the LogP model [6]. Different work has been done to to refine this model, for example taking into account hardware support for long messaging, and to capture memory hierarchies [7, 8]. A more recent parallel machine model for manycores, which considers different core granularities and on-chip and off-chip communication is used in Simplefit [9]. However, this model does not include a fine-granular cost model of tightly-coupled

communication. Taylor et al. propose the AsTrO taxonomy for fine-grained modelling and comparison of scalar operand networks [10].

We propose a machine model, based on the AsTrO taxonomy, which allows a more fine-grained modelling of the overhead associated with communication. The machine model comprises a set of parameters describing the computational resources and a set of timing expressions which describe the computational performance.

2.2.1 Machine Parameters

We assume that cores are connected in a mesh structure. Further that each core to has individual instruction decoding capability and software managed memory load- and store functionality, to replace the contents of core local memory. Such a manycore architecture M is described by the parameter tuple

$$M =< (x, y), p, m, b_g, l_g, o, s_o, s_l, n_b, c, nhl, r_l, r_o >$$

where

- (x, y) is the number of rows and columns of cores.
- p is the processing power of each core, in *operations per clock cycle*.
- m is the size of each cores local memory, in *words*.
- b_g is global memory bandwidth, in *words per clock cycle*
- l_g is the global memory access latency, in *clock cycles*
- o is software overhead for generating a message, 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*
- n_b is the network input- and output buffer capacity, in *words*.
- c is the bandwidth of each interconnection link, in *words per clock cycle*.
- nhl 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.

We assume that the local memory bandwidth is perfectly balanced to the processing power.

2.2.2 Timing Expressions

We now discuss some of the more important timing expressions we use for evaluating parallel computation and communication times on the machine model. Explanations of the expressions are given after each definition when needed.

Computation time The time required to compute the fire code of an actor requiring r_p operations on a core with processing power p is

$$T_p = \left\lceil \frac{r_p}{p} \right\rceil$$

Send time The time required to send r_{s_i} words of data through channel i is

$$T_s = \begin{cases} \left\lceil \frac{r_{s_i}}{framesize} \right\rceil \times o + r_{s_i} \times s_o + T_{sb} & \text{if via the network} \\ \left\lceil \frac{r_{s_i}}{p} \right\rceil & \text{if via local memory} \end{cases}$$

For connected actors that are mapped on the same core, we can choose to map channels in local memory. For connected actors placed on different cores, we can chose to either map channels as buffers in global memory or as a point-to-point stream over the network.

In any case, we assume that data transfers is mapped as a stream of messages. If the maximum message size is *framesize*, we must use $\left\lceil \frac{r_{s_i}}{framesize} \right\rceil$ messages to send the data. The core overhead for generating headers for all messages is $\left\lceil \frac{r_{s_i}}{framesize} \right\rceil \times o$. Further, if a core needs to be active in moving the data to the network, we must add a send occupancy of $r_{s_i} \times s_o$. Finally, if the workloads of the sending and receiving actors are unbalanced, we must take a possible send blocking time, T_{sb}, into account. See Definition 2.2.2.

For channels mapped in core local memory, the send overhead is simply $\left\lceil \frac{r_{s_i}}{p} \right\rceil$ write operations.

Receive time The time required to receive r_{r_j} words of data through channel j is

$$T_r = \begin{cases} \left\lceil \frac{r_{r_j}}{framesize} \right\rceil \times o + r_{r_j} \times r_o + T_{rb} & \text{if via the network} \\ r_{r_j} & \text{if via local memory} \end{cases}$$

The receive overhead is evaluated exactly as the send overhead, except from that parameters of the receiving core have replaced the parameters of the sending core.

Send blocking time Let k be the number of words that can sent until a channel buffer overflows. Let t_b be the estimated blocking time for the send operation for each of the $r_s - k$ blocked words. The send blocking time during the firing of an actor is

$$T_{sb} = \frac{t_b \times (r_s - k)}{r_s}$$

Unbalanced workloads of actors that are communicating might add a blocking overhead due to buffer overflow or underflow. Consider send rate $s_{rate} = \frac{r_s}{T_p}$ and receive rate $r_{rate} = \frac{r_r}{T_p}$ as the average rates, in words per operation, by which tokens are produced and consumed via a channel. If $s_{rate} < r_{rate}$, clearly there will be an underflow of tokens from the sending actor causing the receiving actor to block. Similarly, if $s_{rate} > r_{rate}$ the sending actor might block due to a buffer overflow; depending on the number of words to be sent and the channel buffer capacity. We can calculate the state of buffer to find out whether there will be a buffer overflow, and if so, how many words can be sent until it overflows. Equation (1) calculates the state of a buffer as a function of time t .

$$b(t) = \left\lfloor \frac{r_{s_i}}{T_{p_i}} \times t \right\rfloor - \left\lfloor \frac{r_{r_j}}{T_{p_j}} \times t \right\rfloor + b(t-1) \tag{1}$$

By normalising time to the send rate, s_{rate}, as shown in Equation (2), we get $b(k)$ as a function of sent words; $b(k)$ is the buffer status after k sent tokens.

$$b(k) = k - \left\lfloor \frac{r_{r_j} \times T_{p_i}}{T_{p_j} \times r_{s_i}} \times k \right\rfloor + b(k-1) \tag{2}$$

If $b(k) > n_b$ for $0 < k < r_{s_i}$, the channel buffer will overflow and cause send blocking after k sent words. If the estimated number of blocked cycles per word is t_b, the total blocking time is $t_b \times (r_s - k)$. We distribute the blocking time over each one of all the sent tokens, by computing the average blocking overhead of r_s words.

Receive blocking time Let t_b be the estimated blocking time for a receive operation of each of the r_{r_j} words. The receive blocking time during the firing of an actor is

$$T_{rb} = t_b \times r_{r_j}$$

In the case of an underflow, we simply multiply the estimated blocking time by the number of tokens received (the buffer is empty and the receiver will block at first word).

Network propagation time Let N be the size of a message stream in words. Let d be the distance in number of network hops. Assuming a contiguous message stream, the propagation time is

$$T_c = \begin{cases} s_l + d \times nhl + \left\lceil (N-1) \times \frac{1}{c} \right\rceil + r_l & \text{if using lazy send (DMA)} \\ s_l + d \times nhl + \left\lceil \frac{1}{c} \right\rceil + r_l & \text{if using streamed send} \end{cases}$$

Modelling communication accurately is very difficult. In our model, we assume that SDF graphs are mapped so that the network is contention free.

On machines enabling block transfer operations (DMA) over the network, entire blocks of data can be moved while the core can be utilised for other computation. We use the terminology of *lazy send* and *lazy receive* for this type of transaction. On other machines, cores directly read from or write to global memory via the network. We call this *streamed receive* and *streamed send*.

When using *lazy send*, we assume the whole block of size N in a message must be received before it can be computed on by the receiving core. Network

injection- and extraction latencies is captured by s_l and r_l respectively. We add a delay for each network hop, which is expressed by $nhl \times d$.

The difference between *lazy send* and *streamed send*, is how we account for the network throughput. Depending on the link bandwidth c, when we use *lazy* send, the complete message has been received after $N-1 \times \frac{1}{c}$ further clock cycles. When we use *streamed send*, we only take the head of the stream into account, $\frac{1}{c}$.

Global memory transaction time Let N be the size of a message stream in words. Let d be the distance in number of network hops. Assuming a contiguous message stream, the propagation time for a message stream from or to global memory is

$$T_{mg} = \begin{cases} (N-1) \times max(\frac{1}{c}, \frac{P}{b_g}) + nhl \times d + r_l & \text{if lazy receive} \\ (N-1) \times max(\frac{1}{c}, \frac{P}{b_g}) + nhl \times d + s_l & \text{if lazy send} \\ max(\frac{1}{c}, \frac{P}{b_g}) + nhl \times d + r_l & \text{if streamed receive} \\ max(\frac{1}{c}, \frac{P}{b_g}) + nhl \times d + s_l & \text{if streamed send} \end{cases}$$

We consider global memory transactions as either *lazy* or *streamed* communication over the network. For *lazy receive* and *lazy send*, the first word of the message reaches the end point after $nhl \times d$ time. A latency of s_l or r_l is added to account for the switch to core latency, depending on if the transaction is a receive or a send. The throughput of a message stream is limited either by the link bandwidth c or the number of cores P sharing the bandwidth b_g to a shared memory. Note the reversed proportionality, we chose the maximum value and add a latency of $(N-1) \times max(\frac{1}{c}, \frac{P}{b_g})$ time units to receive the tail of the message.

A *streamed* global memory transaction is modelled almost identically to a *lazy* transaction, except for the absence of the throughput induced latency of the $N-1$ tail words, since the message data can be accessed word by word, once the header has been received.

3 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.

3.1 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 timing expressions introduced in Section 2.2.2. 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.

3.1.1 Vertices.

We distinguish between two types of vertices in G_M^A: *memory* vertices and *core* vertices. Introducing *memory* vertices allows us to represent buffers allocated in 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 an ordered list 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 timing expression T_r, representing the time for an actor to receive data through a channel. The delay of a *compute* operation corresponds to 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 timing expression T_s, representing the time for an actor to send data through a channel.

For a *memory* type of vertex, we simply assign a constant value of the memory access time, specified by l_g in the machine model.

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 list S for each input and output edge respectively.

3.1.2 Edges.

Weighted 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 in local memory, as a point-to-point message stream over the network, or in shared memory using a *memory* vertex.

In the first case we assign $w = T_c$. 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 *memory* actor. We illustrate this by Figure 1. We assign a delay of T_{mg} to the input and output edges to the *memory* vertex.

Figure 2 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

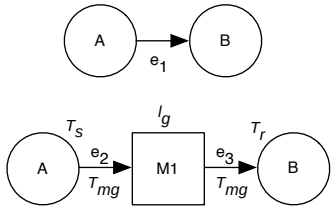

Figure 1: The lower graph (G_M^A) in the figure illustrates how the unmapped channel e_1, connecting actor A and actor B, in the upper graph (A), has been transformed and replaced by a global memory actor and edges e_2 and e_3.

schedule should be interpreted as: actor A fires 6 times, actors B, C and D fires 3 times, and actor E 1 time. The firing of A repeated infinitely by this schedule.

4 Interpretation of Timed Configuration Graphs

In this section we present our model for abstract interpretation. We have implemented an interpreter by this model 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 processes fires 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. In difference to a Kahn process network, PN channels have bounded buffer capacity, which further means that process also temporarily blocks when attempting to write to a buffer that is full [12]. This distinction makes it possible to easily model single message link occupancy on the network.

4.1 Parallel Interpretation using Process Network

Each of the core and memory vertices of G_M^A is assigned to a process. Each of the core processes have a local clock, t, which iteratively maps the absolute start and stop time to each operation in the operations list S. The operations list S is a flattened representation of the hierarchical SDF subgraphs of A.

A core process evaluates a vertex by means of a state machine. Each

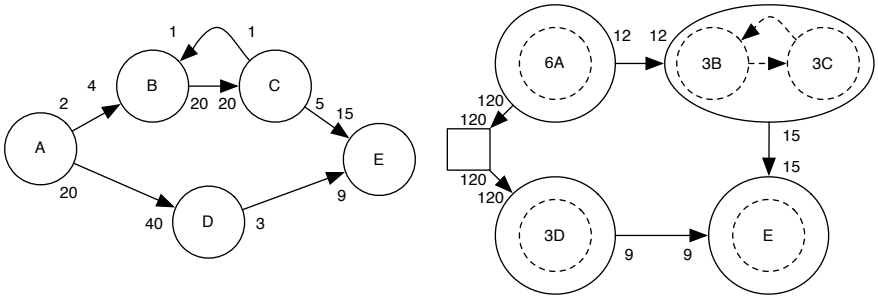

Figure 2: The graph to the right is one possible G_M^A for the graph A to the left. We use dashed lines for actors of A mapped and translated to S inside each core vertex of G_M^A. The feedback channel from C to B is mapped in local memory. The edge from A to D is mapped via a global buffer and the others are mapped as point-to-point message 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.

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 $t = 0$.

Unlike the core processes, the buffer processes do not have a clock. We simply add a read or write delay to the time stamp of a read and write operation and add these events in the *history*.

4.2 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.

4.3 States

For each vertex, we record during what segments of time computations and communication operations were issued. For each process, a *state* 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:

$$StateSet = \{receive,\ compute,\ send,\ blocked\_receive,\ blocked\_send\}$$

The value of t_{start} is the start of the time segment corresponding to the currently processed operation, and t_{stop} is the end of the time segment. For *state*s of types *compute*, *receive* and *send*, the time t_{stop} corresponds to $t_{start}+\Delta$, where Δ is the delay bound to the particular operation. For *blocked_send* and *blocked_received*, the time t_{stop} corresponds to $t_{start} + t_{blocked}$.

4.4 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 implements a read and write blocking FIFO queue. It should be noted that each edge in A is represented by a pair of opposite directed edges in G_M^A

4.4.1 Synchronised Receive

Figure 3 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

receive($t_{receive}$)
    $t_{available}$ = get next send event from edge of sending 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} on edge to the sending vertex
    return $t_{blocked}$
end

Figure 3: Pseudo-code of the receive function. The get and put operations block if the event queue of the edge is empty or full, respectively.

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 $t_{blocked} = t_{available} - t_{receive}$.

4.4.2 Synchronised Send

Figure 4 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 read the previous message and thereby also when the edge is available for sending next message. If $t_{send} < t_{available}$, the *send* operation will block t_{block} clock cycles equal to $t_{available} - t_{send}$. 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 edge read event from receiving vertex
    if($t_{send} < t_{available}$)
        $t_{blocked} = t_{available} - t_{send}$
    else
        $t_{blocked} = 0$
    end if
    put send event $t_{send} + \Delta_e + t_{blocked}$ on edge e to receiving vertex
    return $t_{blocked}$
end

Figure 4: Pseudo-code of the send function. The value of Δ_e corresponds to the delay of the edge.

4.5 Vertex Interpretation

Figure 5 lists the pseudo code for interpretation of a vertex in G_M^A. 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 list S is a process local variable, 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. All states are generated using state generating functions. A state generating function takes timing parameters as input and returns a $state \in StateTypes$. As discussed in Section 4.4.1, the *send* and *receive* functions are the only blocking functions that 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*.

5 Discussion

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

In this paper we have presented our achievements towards the building of a model based manycore mapping and tuning 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 timing expressions 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 benchmarking experiments in order to 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 auto-tuning methods, using feedback information from the evaluator to 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.

References

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

[2] Lee, E.A., Messerschmitt, D.G.: Static Scheduling of Synchronous Data Flow Programs for Signal Processing. IEEE Transactions on Computers (January 1987)

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

[4] Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (Dec 2006)

[5] Taylor, M.B., Lee, W., Miller, J., Wentzlaff, D., Bratt, I., Greenwald, B., Hoffmann, H., Johnson, P., Kim, J., Psota, J., Saraf, A., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., , Agarwal., A.: Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In: Proc. of Int'l. Symposium on Computer Architecture, Munchen, Germany (2004) 2–13

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

[7] Alexandrov, A., Ionescu, M.F., Schauser, K.E., Scheiman, C.J.: LogGP: Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation. In: SPAA. (1995) 95–105

[8] Reif, J.H.: Models and Resource Metrics for Parallel and Distributed Computation. In: Proceedings of the 8th Int'l Symposium on Parallel Processing, Washington, DC, USA, IEEE Computer Society (1994) 404

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

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

[11] Bengtsson, J.: A Model Set for Manycore Performance Evaluation Through Abstract Interpretation of Timed Configuration Graphs. Technical Report IDE0850, School of Information Science, Computer and Electrical Engineering (2008)

[12] Parks, T.M.: Bounded Scheduling of Process Networks. PhD thesis, EECS Department, University of California, Berkeley, Berkeley, CA, USA (1995)

```
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) on edge of subject
                if(t_blocked > 0)
                    add state generateBlockReceiveState(t, t_blocked) to history
                    set t = t + t_blocked
                end if
                add state generateReceiveState(t, Δ of curr_oper)
            end if

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

            else if(curr_op is a Send operation)
                set t_blocked =  value of send(t) on edge of subject
                if(t_blocked > 0)
                    add state generateBlockSendState(t, t_blocked) to history
                    set t = t + t_blocked
                end if
                add state generateSendState(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 5: Pseudo-code of the interpretVertex function.