2009 IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS IX),
Samos, Greece, July 20-23, 2009.

Manycore Performance Analysis using Timed Configuration Graphs

Jerker Bengtsson
Centre for Research on Embedded Systems
Halmstad University
Halmstad, Sweden
Email: jerker.bengtsson@hh.se

Bertil Svensson
Centre for Research on Embedded Systems
Halmstad University
Halmstad, Sweden
Email: bertil.svensson@hh.se

***Abstract*—The programming complexity of increasingly parallel processors calls for new tools to assist programmers in utilising the parallel hardware resources. In this paper we present a set of models that we have developed to form part of a tool which is intended for iteratively tuning the mapping of dataflow graphs onto manycores. One of the models is used for capturing the essentials of manycores that are identified as suitable for signal processing and which we use as target architectures. Another model is the intermediate representation in the form of a timed configuration graph, describing the mapping of a dataflow graph onto a machine model. Moreover, this IR can be used for performance evaluation using abstract interpretation. We demonstrate how the models can be configured and applied in order to map applications on the Raw processor. Furthermore, we report promising results on the accuracy of performance predictions produced by our tool. It is also demonstrated that the tool can be used to rank different mappings with respect to optimisation on throughput and end-to-end latency.**

I. Introduction

For efficient handling of the programming complexity of manycore processors, *domain specific development tools are needed*. One concrete example is the signal processing required in radio base stations (RBS), which is naturally highly parallel and described by computations on streams of data [1]. Many user 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 flexible high-performance processing platforms is manycores. However, there are many issues regarding development of complex signal processing software for manycore hardware. One such is the need for tools that reduce the programming complexity and abstract hardware details of a particular manycore processor. We believe that, if industry is to adopt commercial-off-the-shelf (COTS) manycore technology, the application software, the tools and the programming models need to a high degree be portable.

Research has produced specialised compiler techniques for programming languages based on streaming models of computation, achieving good speedup and high throughput for parallel benchmarks [2]. However, even though a compiler can generate optimised code, the programmer is typically left with very little control of how the source program is transformed and mapped on the cores. This means that, if the code output does not meet the non- functional requirements of the system, the only choice is to try to restructure the source program. We argue that in order to increase performance gain experienced application programmers must be able to control the parallel mapping strategy.

We are developing an iterative code mapping tool that allows the programmer to tune a mapping by:

- analysing the result of a parallel mapping using interpreted performance feedback
- giving timing, clustering and core allocation constraints as input to the tool

Figure 1 outlines the modular architecture of our tool. The tool is designed for using well defined dataflow models of computation. One special case of dataflow, synchronous dataflow (SDF), is very suitable for describing signal processing flows [3]. It is also a good source for code generation to parallel hardware, because it has a natural form of parallelism that is a good match to manycores. The programmer provides a manycore machine specification (using our machine model) and the program (using SDF) as input to the tool. During the model analysis stage, the tool will analyse the processing requirements of the SDF model. As the second stage, we compute a static dataflow schedule for the SDF graph (given that the SDF model is consistent). The scheduled graph is then passed through a model transformation. During the model transformation, the tool generates a timed intermediate representation, which represents an abstract mapping of the application on a specific target processor. We call our intermediate representation a *timed configuration graph*.

In this paper we present our achievements on the models and the timed intermediate representation used by the tool to compute performance feedback to the user. The interpreted performance feedback enables a programmer to, early in the development process, explore the run time performance of the software and to find successively better mappings. We believe that this iterative, machine assisted workflow, is advantageous 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. More

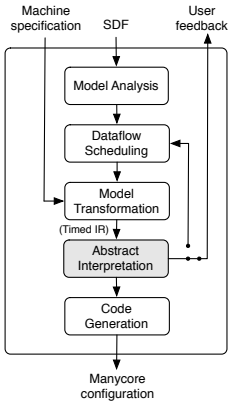

Fig. 1. Outline of the manycore code mapping tool.

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 III, and in Section V we demonstrate the configuration of it for modelling the Raw processor[4].
- An intermediate representation (IR), used to describe a mapping of an application on a particular manycore in the form of a *timed configuration graph*. The use of this IR is twofold: We can perform an abstract interpretation that gives us feedback of performance during execution of the system. Also, we can use it to generate target code. We present the IR in Section IV.
- We make an evaluation of the accuracy and the usefulness of our tool in Section VI. It is shown that our tool is able to correctly rank different mappings of a graph by highest throughput or shortest end-to-end latency.

We conclude the paper with a discussion of the results of the evaluation and we point to improvements in order to increase the accuracy of some of the predictions.

II. Related Work

The problem of mapping task graphs in the form of acyclic precedence graphs (APG) to a parallel processor has been a widely addressed problem. Heuristic solutions are required since this is for a long time known to be an NP complete problem [5]. Sarkar introduced the two step mapping method, where clustering is performed in a first step independently from the second step of scheduling and processor allocation, which can be applied at compile time [6]. A number of leading algorithms, for both single step and two step clustering and merging, with objectives of transforming and mapping task graphs for multiprocessor systems are reviewed in [7].

The dynamic level scheduling algorithm proposed by Sih and Lee is an heuristic taking inter-processor communication overhead into account during clustering. Similar to our work, this scheduling algorithm can be used to produce feedback to the programmer for iterative refining of the task graph and the architecture [8]. However, it has been demonstrated by Kianzad and Bhattacharyya that two step methods tend to produce more qualitative schedules than single step methods [7]. Unfortunately, expanding an SDF graph to an acyclic precedence graph – which are the assumed representation for many scheduling and mapping algorithms – can lead to an explosion of nodes. This problem can partly be reduced using clustering techniques before the SDF graph is transformed to an APG [9]. However, we are interested in techniques for analysis and mapping of SDF graphs without conversion to an APG

The StreamIt language implements a restricted set of SDF. The StreamIt compiler implements a two phase mapping (dataflow scheduling and clustering, followed by core allocation) using direct representation of SDF graphs [2][4]. However, the StreamIt compiler uses a static and location independent cost model for clustering and core allocation. Further, neither the language nor the compiler provides any means to express non-functional constraints or other application specific optimisation criteria to tune the parallel mapping and code generation. Programs have to be restructured in attempts to improve a mapping.

Throughput is one important non-functional requirement in the real-time applications we are addressing. Ghamarian et al. provide methods for throughput using state space analysis on direct representation of multi-rate SDF graphs [10]. Further, Stuijk et al. have developed a multiprocessor resource allocation strategy for throughput constrained SDF graphs [11]. We are addressing techniques that allow combinations of timing constraints and show how to use them to direct the mapping process.

Bambha and Battacharyya provide a good review of different intermediate representations for different objectives on optimisation and synthesis for self-timed execution of SDF programs to multiprocessor DSP systems [12]. They assume homogenous representation of SDF graphs, which exposes a higher degree of task parallelism based on the rate signatures. Our work is similar, but we are mainly interested intermediate representations on multi-rate SDF and in minimising transformation between different representations during the mapping process.

III. 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.

A. Application Model

We model an application using multi-rate SDF. An SDF graph constitutes a network of actors – atomic or composite of variable granularity – which compute on data distributed via synchronous unidirectional channels. Each channel input and output of an actor has an a priori specified token consumption and production rate. By definition, memory and computations in an SDF graph are distributed, and actors 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 periodical repetition schedule. A periodical repetition schedule specifies in which order and how many times each actor fires. If a repetition schedule exists, buffer boundedness and deadlock free execution is guaranteed. One significant advantage with SDF is that the execution order can be determined at compile-time. This enables generation of compact code and elimination of run-time scheduling overhead [13]. The properties of SDF and the formal theory for scheduling of SFD graphs are in detail described in [3].

The Ptolemy modelling framework provides an excellent basis for implementing SDF analysis and code generation tools [14]. Besides serving as input to a code generator, the application model is an executable specification. However, for our work it is not the correctness or the functional properties of the application that is in focus. We are interested in techniques for analysing the non-functional properties of the system. For this we rely on measures like worst case execution time, communication and memory requirements. We assume that these data have been analysed and that each actor is associated with a tuple

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

where

- r_p is the worst case execution time, in number of operations.
- r_m is the requirement on memory 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 where r_{r_j} is the number of words consumed on channel j each firing.

B. Machine Model

Scheduling and core allocation algorithms need to take inter processor (core) communication into account to provide realistic cost measures. These costs in general comprise a static cost for sending and receiving and a dynamic cost determined by the resource location and/or the amount of data to be communicated. However, for reasonably near clock-cycle accurate modelling of dynamic network behaviour it is necessary to use a fine grained cost model for communication. We discuss this further in conjunction with our experimental results in Section VI.

One well-studied and reasonably realistic model for distributed memory multiprocessors is LogP [15]. During the past, much work has been done to refine this model, for example taking into account hardware support for long messaging [16], and capturing network contention [17]. A more recent parallel machine model targeting fine-grained and large scale multicores is developed as a part of the SimpleFit framework [18]. SimpleFit considers variable core granularities and requirements on on-chip and off-chip communication. However, it 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 modelling the 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 [19]. This taxonomy includes a five parameter tuple for comparing and evaluating performance sensitivity of on-chip scalar operand networks.

We propose a manycore machine model based on SimpleFit and the AsTrO five parameter tuple. This model allows a fairly fine-grained modelling of performance, including the overhead of operations associated with communication and off-chip resources. The machine model comprises a set of parameters describing the computational resources and a set of abstract performance functions, which describe the performance of computations, communication and memory transactions.

We assume that cores are tightly coupled via a mesh network. Further that each core has individual instruction sequencing capability and that transactions between core private and shared memory is software managed. The resources of such an abstract manycore architecture are described using two tuples, M and F. M consists of a set of parameters describing the 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 common functions describing the performance of computations, global memory transactions and local communication as functions of M:

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

where

- t_p is a function evaluating the time to compute a sequence 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 specific manycore processor is modelled by giving values to the parameters of M and by defining the functions $F(M)$.

IV. Manycore Intermediate Representation

In this section we describe the 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 program that is transformed 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 predict performance and 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 by an auto-tuner for suggesting a better mapping.

A. Relations Between Models and Configuration Graphs

A timed configuration graph $G_M^A(V, E)$ describes a single connected SDF graph A, transformed and mapped on the abstract machine described by the pair of tuples (M, F). The set of vertices is a union $V = P \cup B | P \cap B = \emptyset$, where P is the set of cores and B is the set of off-chip shared memories. We use v_p to denote a vertex of core type and v_b to denote a vertex of memory type. Edges $e \in E$ are dataflow channels mapped onto the interconnection network of (M, F). To obtain a G_M^A, the vertices of A are clustered with respect to the integrity of the dataflow. Each cluster is assigned to a core in M. The edges of the SDF that end up in one cluster 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 clusters can be implemented in two different ways:

1) as network connection between the two cores. Such connection is represented by an edge (v_{p_i}, v_{p_j}) in G_M^A
2) as a buffer in global memory. In this case, a vertex v_{b_k} is introduced. Further the edge (v_{p_i}, v_{p_j}) is replaced by a pair of edges (v_{p_i}, v_{b_k}) and (v_{b_k}, v_{p_j}) between the two cores in G_M^A.

When G_M^A has been constructed, each $v_p, v_b \in V$ has been assigned costs for computation and communication, calculated using the machine description (M, F) described in Section III-B. These costs reflect the relative costs for each specific operation when computing A on (M, F). We will now discuss how we use A and M to construct and assign costs to the vertices, the edges and the computation costs of G_M^A.

1) Vertices.: Memory vertices, B, allow us to represent a set of buffers mapped in shared memory. A memory vertex can be specified by the programmer, for example to store initial data. Memory vertices can also be automatically generated.

For core vertices, P, 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 cost for a *receive* operation depends on whether the source is another core or a shared memory. Let the source vertex of channel e be $source(e)$. Then for each incoming edge of a vertex p we add a receive operation with a cost bound to:

- $t_r \in F(M)$, if $source(e)$ is of type v_p
- $t_{gr} \in F(M)$, if $source(e)$ is of type v_b

The cost for a *compute* operation is calculated using the performance function t_p, which represents the time required to execute the computations of an actor when it fires.

Finally, for each outgoing edge of a vertex p we add a *send* operation. Let the sink vertex of channel e be $sink(e)$. The *send* operation has a cost bound to:

- $t_s \in F(M)$, if $sink(e)$ is of type v_p
- $t_{gw} \in F(M)$, if $sink(e)$ is of type v_b

Read and write requests on memory vertices are served by the first come first served policy. For a vertex v_b we assign read and write costs calculated using $g_r \in M$ and $g_w \in M$, to account for memory read- and write latencies when serving an incoming request.

When constructing G_M^A, multiple channels sharing the same source and destination can be orderly merged and represented by a single edge, treating them as a single stream of data.

2) Edges.: The weight w of an edge $e(v_i, v_j)$ corresponds to the link propagation. The value of the weight w corresponds to the value of the function $t_c \in F(M)$. Further, edges in SDF can be specified with a sample *delay*. Given an edge $e(v_i, v_j)$, a *unit delay* is defined to mean that the nth sample consumed by v_j corresponds to the $(n-1)$th sample produced by vertex v_i [3]. An edge delay is simply represented by a buffer offset value, needing no further treatment when constructing G_M^A.

Figure 2 shows an example of a simple SDF graph, A, after it has been transformed to one possible G_M^A. One static firing schedule for A in this example is 3(2abcd)e. The 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 indefinitely by this schedule. Thus, no runtime scheduling supervision is required. The feedback channel from actor c to actor b is buffered in core local memory. The edge from actor a to actor d is a buffer in shared (off-chip) memory and the others are mapped as point-to-point connections on the network. The integer values represent the send and receive

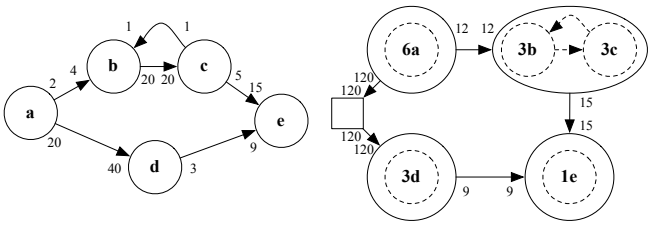

Fig. 2. The graph to the right is one possible graph G_M^A for the application graph A to the left.

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 times an actor fires, as given by the firing schedule.

B. Interpretation of Timed Configuration Graphs

In order to implement and interpret timed configurations graphs, we need a computational model and a notion of time [20]. We have used dataflow process network (PN) to implement interpretable timed configuration graphs [21]. A process network very well mimics the behaviour of the types of parallel hardware we are studying. The PN domain in Ptolemy is a super set of the SDF domain. The main difference in PN, compared to SDF, is that actors are processes which 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 [22]. But, 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. This property enables easy modelling of link occupancy on the network.

Each of the core and memory vertices of G_M^A is assigned to its own process. Each of the 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.

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. Currently, our machine model does not allow modelling of link concurrency. All cores experience the network as a collision free resource. To minimise the risk of providing optimistic performance predictions, we have taken a rather pessimistic approach; only one message is allowed to be sent over an edge during a segment of time, independently of the length of the messages and the network's buffer capacity.

There is no notion of global time in PN. We manage clock synchronisation between the communicating processes by means of communicating *discrete event*s. Send and receive operations generate a discrete event bound to current time. It should be noted that each edge in A needs to be represented by a pair of oppositely directed edges in G_M^A to manage synchronisation. Further, edges in Ptolemy have no ability to perform computations. For each edge, we generate a delay actor, which adds a delay corresponding to the link propagation time ($w \in e \in E$).

V. Modelling the Raw Processor

In this section we demonstrate how we configure the machine model in order to model the Raw processor for performance evaluation [4]. Raw is a tiled, moderately parallel MIMD architecture with 16 (4×4) programmable tiles. Each tile has a MIPS core and is equipped with 32 KB of data and 96 KB instruction caches. The tiles are tightly interconnected via two different types of communication networks: two statically and two dynamically routed.

A. Parameter Settings

We assume a Raw set-up with four off-chip, non-coherent shared memories, and that software managed cache mode is used. Furthermore, we concentrate on modelling the usage of one of the dynamic networks (which are dimension-ordered, wormhole-routed, message-passing types of networks). The parameters of M for Raw with this configuration are set as follows:

$$M =< (4,4), 1, 1, 1, 6, 2, 5, 1, 1, 1, 1, 3 >$$

In our model, we assume a core instruction throughput of p operations per clock cycle. We set $p = 1$. The four shared off-chip DRAMs are connected to four separate I/O ports located on the east-side of the chip. Thus, the DRAMs can be accessed in parallel, each having a bandwidth of $b_g = 1$ words per clock cycle. The latency penalty for a DRAM write is $g_w = 1$ cycle and for a read operation $g_r = 6$ cycles.

The overhead for initiating communication includes sending a header and possibly an address (when addressing any of the off-chip memories). We set the overhead $o = 2$. The four on-chip 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. We have measured and estimated an average send and receive occupancy to be $s_o = 5$ and $r_o = 3$ respectively. Note that we then also include the overhead for reading and writing via buffers in local memory. The network hop-latency on Raw is $h_l = 1$ cycles per router 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 $r_l = 1$.

B. Performance Functions

The performance functions have been formulated by studying the specification of the Raw processor [23].

a) Compute: The time required to process the fire code of an actor on a core is defined 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.

b) Send: The time required for a core to issue a network send operation is defined 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 the Raw 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 (if the local memory capacity is enough). In that case we set t_s to zero.

c) Receive: The time required for a core to issue a network receive operation is defined as

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

d) Network Propagation: Providing means for modeling communication accurately for an abstract parallel target is difficult: high accuracy requires the use of a low machine abstraction level. We chose the approach of modeling communication as collision free.

In the network propagation time, we consider a possible network injection and extraction latency at the source and destination in addition to the link propagation time. The network propagation time is defined as

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

Network injection and extraction latency is captured by s_l and r_l respectively. Further, the propagation time depends on the network hop latency h_l and the number of network hops $d(x_s, y_s, x_d, y_d)$, which is a distance function of the source and destination coordinates. Routing turns add an extra cost of one clock cycle. This is captured by the value of $n_{turns}(x_s, y_s, x_d, y_d)$ which, similar to d, is a function of the source and destination coordinates.

e) 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 memory 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 defined as

$$t_{gr}(r_l, x_s, y_s, x_d, y_d, h_l) = \\ r_l + d(x_s, y_s, x_d, y_d) \times h_l + n_{turns}(x_s, y_s, x_d, y_d)$$

Note that memory read latency penalty is not included in this expression. This is accounted for in the memory model included in the IR (G_M^A).

f) Streamed Global Memory Write: Like the memory read operation, writing to global memory requires two send operations: one for configuring the memory 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

$$t_{gw}(s_l, x_s, y_s, x_d, y_d, h_l) = \\ s_l + d(x_s, y_s, x_d, y_d) \times h_l + n_{turns}(x_s, y_s, x_d, y_d)$$

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

VI. Experimental Evaluation

In this section we present an evaluation of our tool with two purposes:

- to evaluate the accuracy of the tool's performance predictions with respect to actual performance.
- to investigate whether the predictions can be used to rank different mappings of an application with respect latency and throughput.

We have selected two applications with different relations between communication and computation demands to evaluate the accuracy and sources of possible inaccuracy. For the Raw implementations, we have used BEETLE, which is a cycle-accurate Raw simulator.

A. Matrix Multiplication

Our first case study is matrix multiplication, which requires fairly large amounts of data to be communicated over the network. Furthermore, it provides an excellent case for testing the tool on large amounts of communication between the cores and global memory. The input matrices are partitioned into overlapping sub-matrices and the computations are distributed equally on four cores. Thus there is no exchange of data between the cores. Both the input matrices and the result are stored in off-chip memory. Figure 3 shows three different mappings of a 32×32 matrix multiplication used in the experiments. Note that we kept the algorithm the same in all three cases.

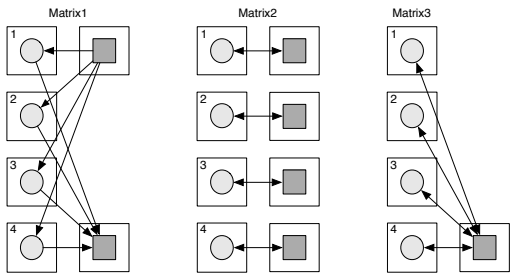

Fig. 3. Three different mappings of the 32×32 elements matrix multiplication using four cores.

In Matrix1, all cores read their assigned input data from the upper memory bank and store the result in the lower memory bank. In Matrix2, we assume that the input data has been arranged and distributed over four separate banks. Thus, in this case, each core has collision-free access to the network and off-chip memory. Finally, in Matrix3, input and output data are all stored in the same memory bank.

We expect the performance prediction for Matrix2 to be more accurate than the predictions for Matrix1 and Matrix3 since our model assumes a collision-free network. Furthermore, by comparing the predictions for Matrix1 with Matrix2 and Matrix3, we expect to get an indication of how sensitive the prediction accuracy is to contention effects. The main difference between Matrix1 and Matrix3 is that, in Matrix3, all communication to the off-chip memory controller is using the same network links. In short, we expect there to be fewer collisions in Matrix1 compared with Matrix3, but the performance should still be relatively close to the performance of Matrix3. This further provides an interesting test case to evaluate whether the tools predictions can be used to determine which mapping performs better.

B. Parallel Merge Sort

Our second case study is merge sort. Compared to matrix multiplication, the merge sort algorithm has very low requirements on computation and communication. Figure 4 shows two different mappings of the merge sort algorithm using 7 and 5 cores, respectively. The computation and communication load, for each vertex in the tree, increases with the level as the tree narrows down. Each vertex in the tree consumes a sorted sub-list from preceding nodes via two channels and produces a merged sorted output. The input data is distributed over the leaf vertices, and the result, a sorted list, is stored locally in the root vertex. In the first of the mappings (called Merge) each of the vertices is mapped to one core. This mapping is illustrated to the left in Figure 4. In the second mapping (called Merge fused, shown to the right in Figure 4), the four leaf vertices have been pair-wise clustered in order to obtain an improved load and communication balance compared to Merge.

C. Accuracy of Predicted Core Communication Costs

In the first experiment, we have studied the accuracy of the predicted performance on send and receive operations.

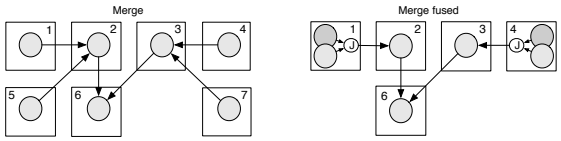

Fig. 4. The graph to the left is a fully parallel mapping of the merge sort (denoted Merge) and in the graph to the right, leaf nodes have been pair-wise clustered and mapped to the same core. The smaller node denoted J, in core 1 and 4, symbolise a join operation performed on the output channels.

For the applications used in the experiments, the programs generated for each core consist of a receive phase, followed by a compute, and then a send phase. We use Raw_{mm} to denote predicted performance (using our tool) and Raw to denote the performance measured on Raw. All predictions and corresponding measurements are made during steady state execution of the dataflow graphs.

Table I shows the predicted receive times, for each used core, compared to the measured receive times for Matrix1, Matrix2 and Matrix3 respectively. The receive time includes possible read blocked time. For each of the three test cases it can be seen that the predicted receive times are slightly pessimistic (which is preferred compared to optimistic). The differences between the predicted and the measured receive times vary between $+2,3\%$ and $+12,6\%$.

In Matrix1, cores 1 and 2 have shorter distance than 3 and 4 to the memory holding the input, which leads to lower read blocking time in the Raw measurements. Because the timed configuration graph views the network as a collision free resource, the receive performance is evaluated more fairly for all cores in Raw_{mm}. Similarly, in Matrix3, cores 3 and 4 have shorter distance to off-chip memory than cores 1 and 2. However, in Matrix3 the unfairness in distance to memory has less importance. Since both read and write request have to compete for the same physical links on the network, the read and write blocking becomes more fairly distributed on the cores.

In the Matrix2 mapping there are no collisions. The main reason for the pessimistic predictions (9,5%) is that we have used averaged measures to configure the send and receive occupancy for Raw. We can probably to some extent tune these parameters to get slightly better accuracy. However, to get a fully accurate prediction we would need to model execution at instruction-level, which would be very costly in terms of modelling performance.

Table II shows the predicted send times compared to the measured send times for Matrix1, Matrix2 and Matrix3. As can be seen, for all three mappings the predicted send time using Raw_{mm} is accurate compared to the measured send time on Raw. The unfairness in distance from the off-chip input memory forces a relative skew between cores during execution (as later explained in section VI-D). Moreover, the send phase comprises much fewer messages to be sent, compared to the receive phase: there are simply no (or very few) collisions during send.

TABLE I
MATRIX STEADY STATE RECEIVE TIME (CLOCK CYCLES)

Application	Core ID	Raw_{mm}	Raw	diff.
Matrix1	1	1790	1589	+12,6%
	2	1790	1589	+12,6%
	3	1790	1750	+2,3%
	4	1790	1750	+2,3%
Matrix2	1	1600	1461	+9,5%
	2	1600	1461	+9,5%
	3	1600	1461	+9,5%
	4	1600	1461	+9,5%
Matrix3	1	1828	1701	+7,5%
	2	1814	1626	+11,6%
	3	1800	1716	+4,9%
	4	1786	1716	+4,1%

TABLE II
MATRIX STEADY STATE SEND TIME (CLOCK CYCLES)

Application	Core ID	Raw_{mm}	Raw	diff.
Matrix1	1	408	408	0%
	2	408	408	0%
	3	408	408	0%
	4	408	408	0%
Matrix2	1	408	408	0%
	2	408	408	0%
	3	408	408	0%
	4	408	408	0%
Matrix3	1	408	408	0%
	2	408	408	0%
	3	408	408	0%
	4	408	408	0%

We will now discuss our corresponding experiment on send and receive times for the merge sort application. In this experiment only core-to-core communication is utilised and the communication consists of very small messages (1 to 4 words). Furthermore, we have deliberately designed one of the mappings (Merge) to force unbalanced core communication and computation loads. This experiment is expected to give an indication on how accurately Raw_{mm} models short messaging and unbalanced communication. The predicted send times compared to the measured ones can be seen in Table III. For Merge, the predicted times are exact or very accurate. Cores 1,2,4, and 7 compute the leaf vertices, which also generate the input in the parallel merge tree. Thus, no receive operations are issued by these cores. However, for Merge fused, we see that Raw_{mm} has evaluated the receive time 75% higher, compared to the measurements on Raw for cores 2 and 3. The reason is that the computation times for cores 2 and 3, after the clustering, are now shorter than for the preceding leaf vertices. Since Raw_{mm} models communication pessimistically – in the sense that we only allow one message at a time on a network link – communication is tighter synchronised in our model. This can introduce blocking times in communication between cores with unbalanced workloads, which are not experienced on Raw.

In Table IV we compare send times for the two different mappings of the merge sort algorithm. As can be seen in the table, the predicted send times are fairly close to the measured times (the difference is 9,1% or less).

TABLE III
MERGE STEADY STATE RECEIVE TIME (CLOCK CYCLES)

Application	Core ID	Raw_{mm}	Raw	diff.
Merge	1	0	0	+0%
	2	16	16	+0%
	3	16	16	+0%
	4	0	0	+0%
	5	0	0	+0%
	6	29	28	+3,6%
	7	0	0	+0%
Merge fused	1	0	1461	+9,5%
	2	42	24	+75%
	3	42	24	+75%
	4	0	0	+0%
	5	0	0	+0%
	6	29	28	+3,6%
	7	0	0	+0%

TABLE IV
MERGE STEADY STATE SEND TIME (CLOCK CYCLES)

Application	Core ID	Raw_{mm}	Raw	diff.
Merge	1	85	79	+7,6%
	2	22	22	+0%
	3	22	22	+0%
	4	85	79	+7,6%
	5	85	79	+7,6%
	6	0	0	+0%
	7	85	79	+7,6%
Merge fused	1	24	22	+9,1%
	2	22	22	+0%
	3	22	22	+0%
	4	24	22	+9,1%
	5	0	0	+0%
	6	0	0	+0%
	7	0	0	+0%

D. Latency and Throughput Measurements

In the second part of the experiments, we have used the same mappings to compare predicted and measured end-to-end-latency and throughput. We also evaluate whether the predictions, despite potential inaccuracy, can be used to rank the mappings correctly with respect to shortest latency and highest throughput.

The mappings are self-timed, meaning that synchronisation is handled at run-time [24]. Initially, a self-timed graph executes a non-steady state and later, after a number of iterations, converges to a steady-state schedule.

Figure 5 illustrates the dynamic behaviour for the self-timed mapping of the Merge application. Cores computing the upstream actors in the dataflow graph with lower workload finish faster and can proceed with the next iteration of the schedule, as long as the network buffer is large enough to store the produced data. As shown in the figure, core 1 has started its fourth iteration when core 6 begins computing its first iteration. When network buffers are full, a steady state execution is naturally forced.

Figure 6 shows the predicted latencies (in clock cycles), for Merge and Merge fused, compared to the measured latencies

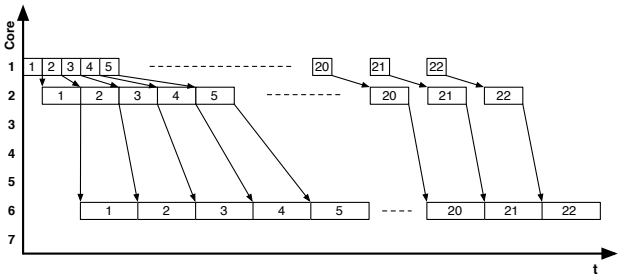

Fig. 5. Skewing experienced in the unbalanced Merge algorithm. The numbers represent the firing count of each actor, and the distance in time between the firings is dependent on the network buffer capacity.

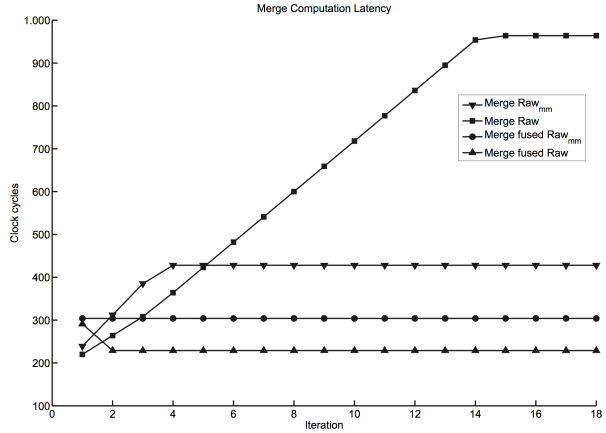

Fig. 6. Comparison of predicted (Raw_{mm}) and measured (Raw) end-to-end latency for Merge and Merge fused.

as a function of the current iteration. The figure shows at which iteration each of the mappings reaches a steady state of execution, i.e. when the latency curve levels out. We see that, for Merge, the measured latency is underestimated by a factor of 2. This is explained by the fact that the machine model is currently not able to model buffer capacity of the on-chip network. Thus, the difference in iteration count between the first upstream actor and the last actor in the graph is larger on Raw than in the modelled execution of Raw. To tighten the latency predictions for graphs with unbalanced communication, we need to account for network buffer capacity in the machine model.

For Merge fused, we see that the latency has rather been overestimated, but is closer to the measured latency. The reason is that both the workload and the communication in Merge fused is better balanced than in Merge (after clustering core 1 with 5 and core 3 with 7), which forces Merge fused to reach a steady state after fewer iterations.

If we rank the predicted latencies of Merge and Merge fused, even if the predictions have varying accuracy, we still see that an optimisation decision based on the predictions would (for this case) correctly identify Merge fused as the better mapping.

Figure 7 shows the predicted end-to-end latencies for

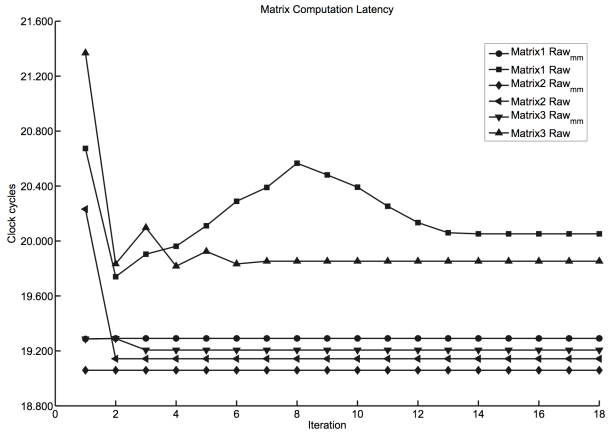

Fig. 7. Comparison of the modelling accuracy of the computation latency of three different mappings of the parallel matrix multiplication.

Matrix1, Matrix2 and Matrix3, compared to the measured latencies on Raw. We see that the different mappings of the matrix multiplication converge to steady state at different numbers of iterations. Unlike in the merge sort experiment, the computation tasks distributed on the cores is naturally load balanced. The reason that the different implementations of the matrix multiplication reach steady state at different points in time is that the cores used in the different mappings are affected by different communication delays due to network contention. Contention effects is a large contributing factor causing an underestimate of the latencies for Matrix1 and Matrix3. This can be verified by observing that the plot for Matrix2 on Raw_{mm} and Raw (which is a contention free mapping), is fairly accurate compared to the predictions for Matrix1 and Matrix3. However, if we rank the predicted steady state latencies for all mappings, we see that an optimisation decision based on latency minimisation would in this case correctly suggest Matrix3 better than Matrix 1 and Matrix2 as the best alternative of the three.

Table V shows the predicted and the measured throughputs for Merge (with 4,4% difference) and Merge fused (with 10% difference). The predictions are fairly close to the measurements on Raw for both Merge and Merge fused. We also see that both the predicted and the measured throughputs show that Merge has a higher throughput than Merge fused. When optimising for throughput, our predictions correctly rank Merge better than Merge fused.

Finally, Table VI shows the corresponding comparisons for Matrix1, Matrix2 and Matrix3. Note that, unlike in all the other experiments, our model has predicted slightly optimistic throughputs. However, if we rank both the predicted throughputs and the measured throughputs, we see that the predictions will be ranked in the same order as for the measured ones. Thus, if using the predictions for throughput optimisation, our tool finds the best cases for this example as well.

VII. Conclusion

In this paper we have presented our achievements on building an iterative manycore code mapping tool. In order

TABLE V
MERGE STEADY STATE PERIODICITY (CLOCK CYCLES)

Application	Raw_{mm}	Raw	diff
Merge	119	104	+4,4%
Merge fused	132	120	+10%

TABLE VI
MATRIX STEADY STATE PERIODICITY IN (CLOCK CYCLES)

Application	Raw_{mm}	Raw	diff
Matrix1	19249	19434	-0,9%
Matrix2	19059	19143	-0,4%
Matrix3	19248	19401	-0,8%

to provide estimates of performance, we have developed a machine model which abstracts a certain category of manycore architectures. We model the applications using synchronous dataflow, and the performance estimates are computed using an executable intermediate representation called *timed configuration graph*.

We have presented an evaluation in terms of the prediction accuracy of our tool and whether the predictions can be used to identify a better mapping. It is shown that communication times between cores are predicted slightly pessimistic, still fairly close to measured performance, with respect to the high level of modelling. Our comparisons indicate that, for the small set of mappings so far explored in the experiments, the tool can correctly rank different mappings with respect to highest throughput or shortest latency. However, the comparisons also reveal that the predictions of end-to-end latency for graphs with unbalanced communication can be quite inaccurate. This was demonstrated to mainly depend on the high abstraction level of on-chip communication implemented by the IR, which currently does not capture the buffer capacity or link concurrency of the network.

To increase the accuracy and the reliability of end-to-end latency measurements on dataflow graphs, we plan to investigate inclusion of network buffer capacity and modelling link concurrency in the intermediate representation. We are especially interested in exploring automatised tuning methods, using feedback information from the abstract interpreter, in order to direct and improve the mapping of application graphs.

ACKNOWLEDGEMENT

The authors would like to thank Dr. Veronica Gaspes at Halmstad University and the anonymous reviewers whose comments helped improving this paper. This work has been funded by research grants from the Knowledge Foundation under the CERES contract.

REFERENCES

[1] E. Dahlman, S. Parkvall, J. Skold, and P. Beming, *3G Evolution: HSPA and LTE for Mobile Broadband*, 2nd ed. Academic Press, 2008.

[2] 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, pp. 152–162.

[3] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Signal Processing," *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24–35, January 1987.

[4] 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*, vol. 22, no. 2, pp. 25–35, 2002.

[5] H. El-Rewini, H. Ali, and T. Lewis, "Task Scheduling in Multiprocessing Systems," *IEEE Computer*, vol. 28, no. 12, pp. 27–37, Dec 1995.

[6] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989.

[7] V. Kianzad and S. Bhattacharyya, "Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 667–680, July 2006.

[8] G. C. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication," Ph.D. dissertation, EECS Department, University of California, Berkeley, CA 94720, USA, April 1991.

[9] J. L. Pino and E. A. Lee, "Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors," in *Proc. of IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing*, 1995, pp. 2643–2646.

[10] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. Theelen, M. Mousavi, A. Moonen, and M. Bekooij, "Throughput Analysis of Synchronous Data Flow Graphs," *Proc. of Int'l Conf. on Application of Concurrency to System Design*, pp. 25–36, 2006.

[11] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal, "Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs," in *Proc. of the 44th annual conf. on Design automation*. New York, NY, USA: ACM, 2007, pp. 777–782.

[12] N. Bambha, "Intermediate Representations for Design Automation of Multiprocessor DSP Systems," in *Design Automation for Embedded Systems*. Kluwer Academic Publishers, 2002, pp. 307–323.

[13] S. S. Battacharyya, "Optimization Trade-offs in the Synthesis of Software for Embedded DSP," in *Workshop on Compiler and Architecture Support for Embedded Systems*, Washington, D.C, 1999.

[14] 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)," EECS Dept., University of California, Berkeley, Tech. Rep. UCB/EECS-2008-28, Apr 2008.

[15] D. Culler, R. Karp, and D. Patterson, "LogP: Towards a Realistic Model of Parallel Computation," in *in Proc. of ACM SIGPLAN Symp. on Principles and Practices of Parallel programming*, May 1993, pp. 1–12.

[16] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. J. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation." in *Proc. of the seventh annual ACM Symp. on Parallel Algorithms and Architectures*, 1995, pp. 95–105.

[17] C. A. Moritz and M. I. Frank, "LoGPC: Modeling Network Contention in Message-Passing Programs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 12, no. 4, pp. 404–415, 2001.

[18] 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*, vol. 12, no. 6, pp. 730–742, June 2001.

[19] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal, "Scalar Operand Networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 16, no. 2, pp. 145–162, 2005.

[20] J. Bengtsson, "A Set of Models for Manycore Performance Evaluation Through Abstract Interpretation of Timed Configuration Graphs," School of IDE, Tech. Rep. IDE0856, 2008.

[21] T. M. Parks, "Bounded Scheduling of Process Networks," Ph.D. dissertation, EECS Department, University of California, Berkeley, CA 94720, USA, 1995.

[22] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. of IFIP Congress 74*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North-Holland Publishing Company, August 5-10 1974, pp. 471–475.

[23] M. B. Taylor, "The Raw Processor Specification," CSAIL, MIT, Cambridge, MA, Tech. Rep., 2003.

[24] E. Lee and S. Ha, "Scheduling Strategies for Multiprocessor Real-time DSP," in *Proc. of IEEE Glob'l. Telecomm. Conf., 1989, and Exhibition. Communications Technology for the 1990s and Beyond.*, Nov 1989, pp. 1279–1283 vol.2.