DC-SIMD: dynamic communication for SIMD processors

Citation for published version (APA):

Frijns, R. M. W., Fatemi, S. H., Mesman, B., & Corporaal, H. (2008). DC-SIMD: dynamic communication for SIMD processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium* (pp. 1-10) <https://doi.org/10.1109/IPDPS.2008.4536274>

DOI:

[10.1109/IPDPS.2008.4536274](https://doi.org/10.1109/IPDPS.2008.4536274)

Document status and date:

Published: 01/01/2008

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

DC-SIMD : Dynamic Communication for SIMD processors

Raymond Frijns, Hamed Fatemi, Bart Mesman and Henk Corporaal
Eindhoven University of Technology
Den Dolech 2, NL-5600 MB Eindhoven, The Netherlands
R.M.W.Frijns@student.tue.nl
{H.Fatemi, B.Mesman, H.Corporaal}@tue.nl

Abstract

SIMD (single instruction multiple data)-type processors have been found very efficient in image processing applications, because their repetitive structure is able to exploit the huge amount of data-level parallelism in pixel-type operations, operating at a relatively low energy consumption rate. However, current SIMD architectures lack support for dynamic communication between processing elements, which is needed to efficiently map a set of non-linear algorithms. An architecture for dynamic communication support has been proposed, but this architecture needs large amounts of buffering to function properly. In this paper, three architectures supporting dynamic communication without the need of large amounts of buffering are presented, requiring 98% less buffer space. Cycle-true communication architecture simulators have been developed to accurately predict the performance of the different architectures. Simulations with several test algorithms have shown a performance improvement of up to 5x compared to a locally connected SIMD-processor. Also, detailed area models have been developed, estimating the three proposed architectures to have an area overhead of 30-70% compared to a locally connected SIMD architecture (like the IMAP). When memory is taken into account as well, the overhead is estimated to be 13-28%.

1. Introduction

The increasing demand for heavy real-time performance under a restricted power budget in media processing applications on mobile devices has pushed the trend in embedded processor design more and more towards parallel architectures. SIMD-type processors are able to exploit the inherent data-level parallelism in those applications; their repetitive structure offers not only efficient and fast processing capacity at a relatively low cost in power, but is also extremely scalable. Especially, image processing applications benefit from the efficiency of SIMD-computing, due to the enor-

mous amount of data-parallelism in pixel-type operations.

Figure 1 shows the major parts of a typical SIMD implementation. It consists of two separate processors, each with a different task during program execution. The control processor controls program flow, checks loop conditions and forwards instructions to the PE-array. The SIMD processor provides the raw computation power by executing instructions received from the control processor in parallel on all its PEs (Processing Elements), while each PE operates on data from its own private memory. The SIMD processor provides status feedback to the control processor, which can be used in calculating branching and loop conditions.

The availability of the cheap, efficient and scalable computational power of SIMD-processing in image applications has motivated surveillance camera manufacturers to implement more and more image processing functionality onto their camera systems, changing the camera into an increasingly 'smart' device (SmartCam, [3]).

However, current SIMD architectures lack support for dynamic communication (i.e. communication over distances that are not necessarily the same for each PE), keeping a whole range of non-linear algorithms (e.g. Lens Distortion Compensation:LDC [6], mirroring, matrix transposition, bucket processing [12]) from being efficiently implemented on SmartCam devices.

Figure 1. Control processor and PE-array of a typical SIMD implementation.

An architecture supporting dynamic communication has been proposed in [11], however in order to be able to function properly, this architecture needs a worst-case amount of buffering for receiving data. This would increase the area overhead to such an extent, that the benefits of dynamic communication support would not justify the cost in extra area anymore.

In this paper, three architectures supporting dynamic communication without the need of large amounts of buffering are presented, each with a different interplay of hardware and software. Communication architecture simulators and area models have been developed in order to study the impact on performance and area of these different architectures.

This paper is organized as follows: related work is shown in section 2, followed by a description of the three architectures in section 3. The architecture simulators and area models are presented in sections 4 and 5. Results of the simulation and area estimations are presented in section 6. Finally, conclusions are made in section 7.

2 Related Work

Several commercial SIMD machines were introduced in the 1980s [9], but they were not widely used. XETAL [1] and IMAP [5] are more recent and interesting SIMD processor examples. They consist of 320 and 256 PEs respectively, arranged as a 1-dimensional linear array. Each PE has only the ability to access its neighbors (left and right) and when it wants to get some data from its n_{th} neighbor ($n > 1$), the corresponding data should be shifted n times to become accessible.

In these architectures, if a particular PE needs to communicate with another PE at a certain distance, all PEs need to communicate with that same distance (due to the SIMD concept). Therefore, they can not efficiently execute applications requiring dynamic communication over variable distances, like LDC. In this application, pixels in a distorted image have to be moved to the right (or left) over variable distances. One way to execute LDC in these SIMDs is to communicate data over the maximum distance (per line) needed. However, this causes severe cycle overhead.

The IMAP processor can handle dynamic communication more efficiently than e.g. the Xetal. Figure 2 shows a schematic of the IMAP communication architecture. Each PE has one shared register where it can send or retrieve data to and from. This shared registers can shift data one position to the left or right every clock cycle. Part of the PE-memory is used as a lookup-table, which is used to check after how many shifts the required communicated data is accessible. In this way, it is possible to use communication over variable distances, however, since the lookup-table has to be filled beforehand, the distances over which data has to be

Figure 2. IMAP communication architecture.

transferred are required to be known at compile-time. Since the lookup table can occupy a large part of PE-memory which can not be allocated for any other use during algorithm execution, the use of it can be considered at a large indirect area cost. Also, the lookup table requires indirectly addressable memory, which is considerably more expensive than a per-line addressable memory.

Imagine [7] is another example of an SIMD which consists of 8 PEs (each PE is a VLIW architecture), where each PE has the ability to get data from all others using a fully connected network between the PEs. If the number of PEs is increased beyond 64 in this architecture (supporting increased data-level parallelism in the architecture), the area related to this communication network will dominate the total area [4]. We conclude that this architecture does not provide a scalable solution.

It seems that an SIMD processor needs either many cycles to perform dynamic communication, or a very rich interconnection structure with a high area cost. The latter possibly also results into high latency and energy consumption. In this paper, we propose new architectures which support dynamic communication over distances calculated at runtime without the additional huge area cost incurred. We compare their performance and area overhead to the IMAP processor, which is selected as a reference architecture due to its ability to handle a restricted form of dynamic communication and because it has, just as the three proposed new architectures, a more expensive indirectly addressable memory system.

3 DC-SIMD architectures

In [11], an architecture concept for dynamic communication support is proposed. This architecture, shown in figure 3, consists of a number of separated buses for left- and right communication, which are segmented by a set of registers. This segmentation is needed to constrain the critical path of the bus network, as well as to allow multiple parallel communications over the different segments. Each PE can write data to all buses, but can only read from one bus.

Figure 3. Architectural concept of dynamic communication support (only left communication is shown).

Since each bus segment is shared by multiple PEs, bus access is arbitrated for, and granted by means of a fixed priority scheme. In [3, 11] it is shown that giving priority to the bus registers (R_i) above the PEs gives the best performance.

Figure 4 shows the message format of data packets on the bus networks. Messages consist of a valid bit, a destination ID and two payload fields. The valid bit is needed to control and indicate the liveness of the message, while the 9-bit (for 320 PEs) destination ID of the message is used by a PE's address comparator to check whether or not the message is intended for that particular PE. The first 16-bit payload field can keep a (pixel) data value, the second 16-bit payload field is used to either keep the storage location of the first payload item (required for lens distortion compensation), or the source ID of the sending PE (required for e.g. FIR filtering, convolution).

The destination ID and data fields of the message can be set by the ALU of the PE; sending involves setting of the valid bit in the message (which is user-initiated and fully programmable). From that point, non-programmable hardware takes control of the message. Upon sensing the valid bit, the bus arbiter will draw it in its arbitration scheme and eventually put the message into a bus register. Thereafter, the message is repeatedly transferred to the next bus register, until it is retrieved by the destination PE. For example, if PE6 wants to send a message to PE1 (fig. 3), it first competes with PE5 and PE7 to get access to bus register R4. After getting access to R4, the message is propagated to R1, where PE1 can retrieve it.

After the transfer from bus register to input buffer, the communication is programmable again; further operations on the received message, i.e. transferring it to the ALU or

Figure 4. Communication messages format.

memory of the PE, is performed in software and is user-initiated.

(a) Original image

(b) Transposed image

Figure 5. Image artifacts in a transposed image caused by buffer overflows in the concept architecture (simulation result).

One of the issues that should be considered with this architecture is input buffer size. A PE must *always* accept a message intended for it at the moment it passes its corresponding bus register, since after it propagates further in the next cycle, the message will not be able to come back to that particular bus register anymore. Therefore, an input buffer is needed to store messages from the bus until they are transferred to the PE itself. However, with dynamic communication the amount of communication is known only at run-time, so the architecture must be able to deal with the worst-case load, which is an all-to-one communication (e.g. a matrix transpose). This would require a buffer size equal to the number of PEs *per PE*, which is unacceptably costly in area.

Reducing this buffer size is paramount for a feasible implementation, however, simply reducing it below the worst-case amount will cause loss of messages at high bus loads, which results in artifacts in the output image (fig. 5), or numerical errors in applications that calculate the statistics of

an image.

Also, the message transfer from input buffer to the ALU or PE-memory can cause the same trouble if not organized properly. The transfer from bus to input buffer is done by hardware, but at some point in time the message has to be transferred to the ALU or memory of the PE. However, a PE does not know in advance *when*, *how many*, or even *if* it will get any messages; so correspondingly, it does not know exactly when and how often to check its input buffer. Checking not often enough, or checking at the wrong moments, will cause image artifacts due to missing pixels. Implementing the check by a blocking conditional, i.e. waiting until a valid message arrives, will resolve the timing uncertainty, however, if the PE will not get any message, it will deadlock. So correctly handling this process is paramount in realizing a feasible implementation of a DC-SIMD architecture.

In the following subsections, three DC-SIMD architectures are presented which solve these issues, each by their own interplay of hardware and software.

3.1 Basic architecture

The basic architecture shown in figure 6 solves aforementioned issues by employing a high bandwidth from bus to PE-memory. In a first step to achieve this, the software-side of the architecture (fig. 6, left) organizes communications into separate blocks by placing the receive process into a blocking loop; after each send instruction, the host processor 'captures' the PEs in this loop by continuously issuing receive instructions until all messages have been received by the PEs. In this way, each communication is completely handled before continuing with the rest of the program. This ensures that *all* messages have been received when the loop exits (resolving the timing issue). In order to maximize the bandwidth to the PE memory, the loop body is restricted to only keep a single receive instruction, so it will be capable of transferring the worst case amount of messages arriving in *one clock cycle* to memory. In this way, only the one-cycle worst-case amount of messages needs to be stored in dedicated area-intensive buffering.

Some PEs receive their message(s) earlier than others, and not all PEs have the same amount of data to send in a single send/receive block. However, this contradicts the SIMD-principle that *all PEs* must execute the *same instruction the same time*. Therefore, the send and receive instruction are both to be implemented as guarded instructions. By setting/resetting its own guard bit *G*, a PE can individually 'switch off' one or more of its guarded instructions. This also improves energy efficiency, since 'executing' disabled instructions cost considerably less energy.

Even though the loop body itself is implemented in software, the exit condition for the loop is implemented in hard-

Figure 6. Basic Architecture.

ware by OR-ing all the valid bits of the bus registers (fig. 6, right) and the send/receive buffers of the PEs. This global OR signal indicates that there are no valid messages in the complete communication pipeline, i.e. all messages have been delivered for that particular communication block. During the receive loop, the host processor keeps sending receive instructions to the PEs while checking the global OR-signal. When the OR-signal is set, the host processor resumes the execution of the program by issuing the next scheduled instruction.

Program 1 shows the pseudo-code of a mirroring application, where pixels are moved horizontally over a distance dependent on the distance to the vertical line through the image center. Code lines preceded by *CP* are executed on the control processor, code lines preceded by *PE* are executed on the PE-array. The PE code lines are executed on all PEs in parallel, for *PE_ID* in the range of 0..319.

Program 1 Pseudo-code for mirroring a 320x240 image running on PE with index number 'PE_ID'.

```

CP:   for (line = 0; line < HEIGHT; line++) {
PE:   message.dst = 320 - PE_ID;
PE:   message.data = memory[line];
PE:   send;
CP:   while (!bus empty) {
PE:   receive @ memory[line];
    }
  }

```

Besides the OR-network, the hardware-side of the architecture consists of the segmented bus network of the concept architecture with adequate buffering to store the messages that can arrive in one cycle. Since each PE has a read port on two buses (left- and right communication), the worst-case message arrival rate per clock cycle is 2, so a receive buffer of that size is needed between bus registers and

The worst-case transfer of two messages to PE-memory in the same clock cycle requires an expensive dual-port memory. Alternatively, a double-width single-port memory could be used, requiring the two messages to be stored at the same double-width memory location. However, during clock cycles in which only one message arrives, half of the double-width memory location is unused, resulting in very inefficient memory utilization. Also, a double-width memory is energy-inefficient compared to a single-width memory.

The basic architecture of the previous section puts a high burden on the memory system, requiring a high bandwidth to memory in order to be able to store two messages in the same cycle. Also, upon the start of each communication block, enough PE-memory should be reserved to store the worst-case amount of messages, or even worse, memory locations could be overwritten if there is not enough memory available. Furthermore, this architecture is not very efficient for applications requiring some calculation on the received messages (e.g. convolution, filtering), since the data is then transferred to and from the memory twice.

One way to implement this flow control, is on a per-register base. Within one clock cycle, a bus register should be able to sense a control signal from the next bus register and/or receive buffer. So upon a full receive buffer, such a control signal can be sent to the corresponding bus register, causing it to stall itself in the next clock cycle. On his turn, this stalled bus register sends a control signal to the previous bus register, which will stall in the cycle thereafter, and so on. In this way, the bus stall ripples like a wave through the network, stalling one new bus register per cycle, all the way to the beginning of the bus. At the moment storage locations are available again at the receive buffer that caused the stall, the control signal is reset, allowing the first stalled bus register to accept incoming messages again. In the same way the stall wave propagates, the 'release' wave also propagates with a speed of one register per cycle through the bus. So, besides the fact that recovering from such a stall wave takes as many cycles as creating it, *each* bus register

Therefore, the stall process must be implemented as a complete simultaneous bus stall. Due to the propagation delay of the control signal network, the stall process will probably take more than one cycle. Therefore, the bus should be stalled *before* a receive buffer is completely full, in order to store possible incoming packets during the multi-cycle stall. The required slack locations equals the number of cycles needed to halt the bus multiplied with the worst-case arrival rate per cycle.

In contrast to the previous architecture, where only the send and receive instructions have to be guarded, this architecture requires a completely guarded instruction set (since any instruction is now allowed in the loop body).

The architectures of sections 3.1 and 3.2 both suffer from cycle overhead caused by the block organization of communications, where each receive is handled in a blocking loop before the program can continue. This is not very time-efficient, since many PEs will spend most of the loop execution time waiting for the message with the longest path to be delivered.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on December 17, 2008 at 06:03 from IEEE Xplore. Restrictions apply.

Figure 8. Pipelined architecture.

handling a complete receive process before proceeding with the rest of the program, i.e. continuing with the next iteration before finishing the receive process of the current iteration.

Now, the send instruction moves a message to the send buffer, and the PE waits until bus access is granted (fig. 8). Due to the priority scheme used to get bus access, each PE exits its waiting loop at a different cycle number, so PEs execute the same instructions in the same order, but at *different moments in time*, i.e. they execute their instructions independent from each other (PEs are not operating synchronized to the *same* clock cycle anymore). This not only reduces the number of waiting cycles of the PEs, but also spreads the communication load over time. However, the order of arrival of messages is not always fixed anymore; since now each receive process is not resolved completely before proceeding with the rest of the program, the send instructions of different image lines will mix up. If, for instance, a PE is to receive a message from a PE far away in one cycle, and again from a PE close by a few cycles later (while the first message is still being propagated through the bus), the second message could be delivered before the first. Therefore, in applications where the order of arrival is of importance, the source ID of the sending PE is placed in the second data field of the message, so the receiving PE can still distinguish the origins of the different received messages.

Because not all PEs fetch the same instruction at the same time, each PE needs an instruction buffer (top right of fig. 8), as well as additional hardware to re-synchronize at the end of an image frame. Due to pipelined execution, the image is not processed line-by-line (as with the other architectures), but each image column is processed independently of the others (each PE has one or more image columns under its control). Since some columns are fin-

ished earlier than others, all PEs have to be synchronized at the end of an image frame. To this end, each PE sets a guarding flag after processing all pixels in its frame column. This guard switches off some of the instructions in the loop, preventing the PE from fetching new data from memory, while still enabling it to receive and store data. The exit condition for the loop structure is satisfied if the AND-ing of all the guarding flags is set and the OR-ing of the valid bits in the bus registers and buffers is reset, i.e. when all PEs have processed and communicated their column of pixel data.

Since with this architecture all instructions are within the same loop body, the receive instructions could occur less frequently than the worst-case arrival rate. Therefore, the bus-stalling functionality of the architecture of section 3.2 is also needed in this architecture to stall the bus in case of a full input buffer. Also, this architecture requires a completely guarded instruction set, since any instruction is allowed in the loop body.

3.4 Instruction Set Architecture

Figure 9 shows the instruction format of the DC-SIMD. The instructions are 24 bit wide, and have a fixed RISC-like format to keep the instruction set simple though powerful enough to efficiently handle most constructs. The instruction format is the same for both CP and PE-array.

Figure 9. Instruction format of DC-SIMD instructions.

In the DC-SIMD architecture the program memory is shared by the CP and PE-array. Since only one instruction is fetched per cycle, this means that in general, either the CP or the PE-array is active. This approach has a drawback of hiding the existing independence between the two processors, but it greatly simplifies the programming task, since the coding is sequential.

A CP-instruction is distinguished from a PE-instruction by its first instruction field, the ID bit. An instruction with a 0 in its ID-field is transformed into a NOP by the PEs, while an instruction with a 1 in its ID-field is transformed into a NOP on the control processor.

As mentioned in section 3, DC-SIMD needs a guarded instruction set, where PEs and the CP can 'switch off' some of their instructions, only executing them when some condition is met. To this end, the guard bit G is used. When the guard bit is set, the instruction is executed unconditionally,

but if the guard bit is not set, it is only executed when a local flag register is set.

The third instruction field, the 6-bit Opcode-field, is used to distinguish between different operations. For the control processor, these operations primarily consist of branching, comparison and program jumps, while the PE operations are more tuned for computation.

The next three 4-bit wide instruction fields are used for addressing the register file (containing 16 registers) and providing an 8-bit immediate value.

The instruction fields are all organized in groups of 8 (ID+Opcode, Immediate) or 4 bits (Register addresses), making it very simple to program in binary code during development, and simplifying the construction of an assembler.

4 Simulation

In order to make accurate assumptions on the expected performance of the three architectures, high-level architecture simulators (in C++) have been developed. In these simulators, the emphasis is put on the behavior of the communication architecture while abstracting from the actual implementation.

Figure 10 shows the simulation model of the DC-SIMD architectures. The bus registers, buffers and data memory of the PEs are modeled as a set of arrays, which can transfer messages between them. The lower part of the figure models the behavior of the bus network, where transfers are automatic every simulated cycle. In parallel, the PE behavior is modeled (upper part of the figure), where transfers are parameterizable (order of actions, how many cycles to execute them).

The actual computation performed by the PEs is not simulated, only the related time delay is accounted for, so simulation is still cycle-true. Computation itself is performed before the actual simulation, storing its results in the message buffer. Simulation involves consuming messages from the message buffer and getting them to the correct PE-memory through consecutive simulation cycles.

5 Area model

For all three architectures, an area model has been developed in order to study the impact of the architectural features on overall area. Figure 11 shows the modeled per-PE area contributions. The different area contributions consist of the PE itself, a bus segment, some send/receive buffering and control structures for bus stalling and frame-synchronization.

The control logic for these features is shown in figure 11(a). To control the number of loop iterations, the CP has

Figure 10. Simulation of the DC-SIMD communication architectures.

to check whether or not all messages have been received by the PEs. To this end, the valid bits of the bus registers, send buffers and the first locations of the receive buffers need to be OR-ed. Per PE, this requires 3 'internal' OR-gates and 1 extra OR-gate to combine it with the result of other PEs in order to make it a global OR.

The bus stall signal should be triggered when one or more input buffers is (nearly) full. This is implemented by OR-ing on some threshold index of the receive buffer. As with the loop control signal, the bus stall signal should be OR-ed with the signal of the other PEs.

The pipelined architecture requires some synchronization after each processed frame. To this end, each PE can set a flag register to indicate that it has processed all its frame data. A global AND on these flags will, together with the result of the global OR, indicate that all PEs are synchronized and ready for the next frame. A single AND-gate on the flag register is sufficient to generate the synchronization signal.

The arbiter and read logic which are part of a bus segment are shown in figure 11(b). Since DC-SIMD has separate buses for both direction, the bus segment of each PE consists of 2 bus registers, 2 address comparators, 2 arbiters and write logic.

The area contributions of these architecture components are shown in eq. 1.

Figure 11. Hardware modeled in the area model.

$$\begin{aligned}
 A_{bussegment} &= 2 \cdot w_{msg} \cdot A_{reg} + w_{dst} \cdot (A_{XOR} + A_{AND}) \\
 &\quad + 2 \cdot (w_{msg} \cdot A_{mux4to1} + A_{prioritylogic}) \\
 &\quad + 2 \cdot w_{msg} \cdot (A_{mux2to1} + 2 \cdot A_{mux3to1}) \\
 A_{send+rec.buf.} &= (N_{buf.loc} + 1) \cdot w_{message} \cdot A_{reg} \\
 A_{instr.buf.} &= N_{instr.buf.loc.} \cdot w_{msg} \cdot A_{reg} \\
 A_{loopcontrol} &= 4 \cdot A_{OR}, A_{busctrl} = A_{OR} \\
 A_{sync} &= A_{AND}
 \end{aligned} \tag{1}$$

In the basic architecture, the per-PE architecture components are PE itself, a receive buffer of size 2, a send buffer of size 1, a bus segment and the OR-network for loop control. The per-PE contribution for the architecture with bus control is equal to that of the basic architecture extended with bus control logic and a larger receive buffer. The pipelined architecture is again extended with an instruction buffer and extra loop control logic for required for synchronization. Adding the relevant architecture components results in the following area model for the three DC-SIMD architectures:

$$A_{Basic} = N_{PEs} \cdot (A_{PE} + A_{bussegment} + A_{send+rec.buf.} + A_{loopcontrol}) \tag{2}$$

$$A_{Buscontrolled} = A_{Basic} + N_{PEs} \cdot A_{busctrl} \tag{3}$$

$$\begin{aligned}
 A_{Pipelined} &= A_{Basic, architecture} \\
 &\quad + N_{PEs} \cdot (A_{busctrl} + A_{sync} + A_{instr.buf.})
 \end{aligned} \tag{4}$$

The IMAP processor performs inter-PE communication by means of a shift-register, where each PE can determine when to pick up data from the bus by means of a large lookup table [10]. In the area model for the IMAP, the lookup-table is not considered, since its size depends on the application. The contributions to the total area are assumed to be dominated by the PE area and the shift register bus, resulting in eq. 5.

$$A_{IMAP} = N_{PEs} \cdot (A_{PE} + A_{reg} \cdot w_{data}) \tag{5}$$

Wiring and the PE-memory have not been taken into account in the area models.

6 Experimental results

In this section, the results from simulation and area estimation are presented and evaluated. First, the architecture simulators are used to explore the effect of buffer size on the performance. From these experiments, a suitable buffer size is chosen for each architecture, which will be used in the performance comparison and area estimation.

6.1 Performance

To benchmark the performance of the different architectures, five test algorithms have been used. *Mirror* mirrors an input image over a vertical line through its center,

while *Transpose* [2] mirrors over the diagonal (matrix transposition). With *LDC*, the non-linear lens distortion is compensated by shifting pixels in a radial direction over a distance dependent on the pixel's distance to the center of the image. *Bucket* is a bucket processing algorithm which collects pixels in different PE memories based on their intensity. Finally, *Convolve5x5* is a convolution with a 5x5 skeleton over the image. *Transpose* and *Bucket* both invoke many-to-one communications, while the other algorithms use only one-to-one communication.

In order to find a suitable input buffer size, all five test algorithms are simulated with different buffer sizes. Figure 12(a) shows the execution time of the algorithms for different buffer sizes on the pipelined architecture. As expected, the algorithms with one-to-one communication (*Mirror*, *LDC* and *Convolve*) show no dependence of the buffer size. The algorithms which use many-to-one communication (*Bucket*, *Transpose*) show stable performance at a buffer size around 4 to 6. This also holds for the architecture with bus control. Therefore, an input buffer size of 4 with 2 extra storage locations to compensate for the delay in the bus-stalling process is chosen for the architecture with bus control and the pipelined architecture, reducing the amount of required buffer space by 98% compared to the architecture concept. The performance of the basic architecture does not increase with extra buffer space, since the worst-case load is already matched by the high throughput to PE-memory, so for this architecture the input buffer size is fixed at a size of two, which is a buffer space reduction of 99% compared to the architecture concept.

Figure 12(b) shows the relative execution times of the different DC-SIMD architectures compared to IMAP. On average, the three architectures execute the test programs 42-55% faster than the IMAP. On the basic and bus control DC-SIMD architectures, the *Mirror* and *Transpose* and *Bucket* test algorithms execute about 65% faster. On the pipelined DC-SIMD architecture, they execute even up to 83% faster. These three algorithms have the highest communication distances, and therefore, DC-SIMD has a the biggest advantage of its faster propagation (due to the multiple-bus architecture). The *LDC* algorithm runs respectively 8% and 25% faster on these architectures, while the *Convolve* algorithm executes in roughly the same number of cycles on all architectures. These algorithms have considerably shorter distance communication, so the faster propagation speed of DC-SIMD has less impact here. Specifically, the *Convolve* algorithm involves only short distance static communication, which traditional SIMD architectures can already handle efficiently.

(a) Performance for different input buffer sizes of the pipelined DC-SIMD architecture.

(b) Performance of different architectures compared to the IMAP processor.

Figure 12. Simulation of different benchmarking algorithms.

6.2 Area

Table 1 summarizes the parameters used in the area model [3]. With these parameters as input for the area model (section 5), the area overhead compared to IMAP for the three architectures is estimated 34% for the basic architecture, 43% for the architecture with bus control, and 71% for the pipelined architecture.

In the area models, data memory (and instruction memory) has not been taken into account. Memory area can easily dominate overall chip area, and since the DC-SIMD processor is still under development, the exact amount of memory has not yet been determined. Incorporating memory area in these early area estimations would result in too much uncertainty. The current area estimations can therefore be seen as a 'worst-case' approximation, since the overhead of the DC-SIMD architectures will considerably lessen when memory area is taken into account. Assuming, like in the Xetal [8], a memory area equal to 1.6 times the area of the PEs, the area overhead for the three architectures already

drops to respectively 13%, 17% and 28%.

Table 1. Parameters used in the area model

Parameter	Value	Parameter	Value
w_{data} (bits)	16	A_{reg}	5
w_{msg} (bits)	42	A_{XOR}	3
w_{instr} (bits)	24	A_{OR}	1.5
w_{dst} (bits)	9	A_{AND}	1.5
		$A_{mux2to1}$	2
$N_{instr.buf.loc.}$	24	$A_{muxNto1}$	$N \cdot \log(N)$
$N_{buf.loc.}$	0...50	$A_{priority}$	7.5
N_{PE}	320	A_{PE}	10000

7 Conclusions and future work

In this paper, three architectures are proposed to support dynamic communication in SIMD-processors. An earlier architecture concept supporting dynamic communication requires large amounts of buffering to work correctly, while the proposed architectures reduce the amount of required buffer space by 98%, by using different interacting hardware/software architectures.

For each proposed architecture, a cycle-true communication architecture simulator has been developed to accurately predict its performance. Simulations have shown an average performance improvement of 41%, 37% and 55% compared to a locally connected SIMD-processor (the IMAP).

Also, a detailed area model has been developed, estimating the area contribution of the various architecture components. The three proposed architectures have a worst-case area overhead of 34%, 43% and 71% compared to IMAP when only the area of computation and communication is considered. However, taking a memory area of 1.6 times the area of the PEs into account, the total area overhead of the three architectures compared to IMAP is only 13%, 17% and 28% respectively.

An instruction set architecture and structural layout have been designed for the DC-SIMD architectures, and currently a low-level assembly-programmable simulator is under development. This simulator is to be followed by an FPGA-implementation, and development of a DC-SIMD compiler.

References

- [1] A. Abbo and R. Kleihorst. Smart Cameras: Architectural Challenges. In *Proceedings of Advanced Concepts for Intelligent Vision Systems (ACIVS)*, pages 6–13, Ghent, Belgium, September 2002.
- [2] W. Bokhove. Fast Robot Vision using the IMAP-VISION Image Processing Board. Master's thesis, Delft University of Technology, Delft, The Netherlands, 2000.
- [3] H. Fatemi. *Processor Architecture Design for Smart Cameras*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2007.
- [4] H. Fatemi, H. Corporaal, T. Basten, R. Kleihorst, and P. Jonker. Designing Area and Performance Constrained SIMD/VLIW Image Processing Architectures. In *Proceedings of Advanced Concepts for Intelligent Vision Systems (ACIVS)*, pages 689–696, Antwerp, Belgium, September 2005. Springer-Verlag, Berlin, Germany, 2005.
- [5] Y. Fujita, S. Kyo, N. Yamashita, and S. Okazaki. A 10 GIPS SIMD Processor for PC-based Real-Time Vision Applications — Architecture, Algorithm Implementation and Language Support. In *In Proceedings of the 4th International Workshop of the Computer Architecture for Machine Perception, (CAMP)*, pages 22–32, Washington, DC, USA, October 1997. IEEE Computer Society.
- [6] P. Jonker, J. Caarls, and W. Bokhove. Fast and Accurate Robot Vision for Vision Based Motion. *Lecture Notes in Computer Science*, 2019:149–158", 2001.
- [7] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, April 2001.
- [8] R. P. Kleihorst, A. A. Abbo, A. van der Avoird, M. O. de Beeck, L. Sevat, P. Wielage, R. van Veen, and H. van Herten. Xetal: a low-power high-performance smart camera processor. In *ISCAS (5)*, pages 215–218. IEEE, 2001.
- [9] D. J. Kuck. A Survey of Parallel Machine Organization and Programming. *ACM Computing Surveys*, 9(1):29–59, 1977.
- [10] S. Kyo. A 51.2 GOPS Programmable Video Recognition Processor for Vision Based Intelligent Cruise Control Applications. *Proceedings of the 2002 IAPR Workshop on Machine Vision Applications*, pages 632–635, 2002.
- [11] B. Mesman, H. Fatemi, H. Corporaal, and T. Basten. Dynamic SIMD for lens distortion compensation. *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, 00:261–264, 2006.
- [12] E. Olk. *Distributed Bucket Processing*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2001.