Performance Modeling and Analysis of Exposed Datapath Architectures

Klaus Schneider, Demyana Selim, and Nadine Kercher
RPTU University Kaiserslautern-Landau
Kaiserslautern, Germany
https://es.cs.rptu.de

2025 Forum on Specification & Design Languages (FDL) | 979-8-3315-9379-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/FDL68117.2025.11165276

***Abstract*—In exposed data path architectures, registers are replaced by an on-chip network that connects their processing units (PUs) directly. This allows the compiler to determine PU allocation, instruction scheduling, and data transport between the PUs. To prevent unnecessary synchronization of the PUs, their network ports are typically buffered. Although many performance models are available for traditional RISC architectures, there are no specific performance models for buffered exposed datapath (BED) architectures.**

In this paper, we investigate the impact of the relevant design parameters of BED architectures, consider their dependencies, and determine reasonable parameter values for designing cost-effective efficient BED processors. In particular, we examine the number of PUs, the instruction issue width (superscalarity), the size of network buffers, and the latency of instructions, and relate these parameters with the processor performance. We develop a performance model to estimate the runtime in terms of the mentioned parameters and validate our performance model with experimental results.

***Index Terms*—exposed datapath architectures, processor performance models, design space exploration**

I. Introduction

Modern processors consist of many concurrent components such as register files, reservation stations, reorder buffers, processing units, branch predictors, and caches that interact closely with each other. The *design parameters* of these components must be determined during processor implementation and affect the overall cost and performance of the processors. However, determining these parameters with hardware prototypes is nearly impossible, and even the use of instruction set simulators is too time-consuming due to the prohibitively large design space.

For this reason, various *performance models* have been introduced [10], [17]–[20], [27], [28], [35]–[37], [39], [44], [45], [47], [49]–[53], [61], [65], [68], [72], [73], [75] for RISC processors. These performance models are not meant to replace instruction set simulators, but rather to narrow the design space by identifying dependencies between their design parameters. Thus, they can guide the use of simulators, so that fewer simulation runs are needed to finally determine design parameters. The available performance models for pipelined and superscalar RISC processors provide a good understanding of the relationships among their parameters.

Unfortunately, this is not the case for more recent types of processor architectures, such as *exposed datapath architectures* such as TTAs [14]–[16], [29], [32], [38], [77], RAW/Tilera [66], [67], [71], TRIPS [12], [23], [55], [56], Tartan [48], DySER [24], FlexCore [69], AMIDAR [22], STA [13], SCAD [2], [5]–[7], [58], [60], Mozart [57], and others [21], [25], [30], [31], [70], to name a few. As their name suggests, exposed datapath architectures expose their internal datapaths to the compiler. This allows the compiler to allocate *processing units (PUs)* for the instructions, to schedule the instructions on the PUs, and to manage the communication of intermediate results between PUs.

Exposed datapath architectures are typically *hybrid dataflow/von Neumann architectures* [8], [11], [34], [74]. Therefore, they still run sequential programs, but locally exploit *instruction-level parallelism (ILP)* through dataflow computing which means that computations are triggered when operands are available. Typically, this requires buffers to store intermediate results until they can be processed further. Hence, many of these architectures are *buffered exposed datapath (BED)* architectures. Their main architectural paradigm is the decentralization of all processor components [54], [76] to achieve quasi-linear scalable circuit designs [12], [55] that can fully exploit the available ILP of programs. Due to their different architecture, BED processors are based on different design parameters than RISC processors.

In this paper, we therefore investigate the impact of the design parameters of BED architectures, consider their dependencies, and argue about reasonable values for implementing cost-effective efficient BED processors. In particular, we examine the number of PUs, the instruction issue width, the size of buffers, and the latency of instructions, and relate these parameters with the runtime of the program to determine their impact on the performance. We develop a performance model to estimate the runtime in terms of the mentioned parameters and validate our performance model with experimental results.

The paper has the following outline: In the next section, we review related work on performance models, and explain their use in processor design. We also describe a general model of BED architectures that we use for our performance model. Section III is the core of the paper and presents the performance model which is validated by experiments in Section IV.

II. Related Work

A. Performance Models

Software simulators of processors such as SimpleScalar [3] are based on the *instruction set architecture (ISA)* and the microarchitecture of the processors. They can be used to quickly evaluate the runtime of given programs for specific values of design parameters before hardware prototypes are actually implemented and can therefore be used to speed up the processor design.

However, although ISA simulators are very fast, the design space of complex processors which depend on many architectural parameters is too large to be fully explored by simulation alone. For this reason, *performance models* [10], [17]–[20], [27], [28], [35]–[37], [39], [44], [45], [47], [49]–[53], [61], [65], [68], [72], [73], [75] have been introduced to identify dependencies between architectural parameters which can be used to narrow the design space. Although these performance models are less precise than ISA simulators, they can be used for a rough design space exploration that can be refined by ISA simulators and hardware prototypes. Several kinds of performance models for processors have been developed which allow us to better understand the use of *instruction level parallelism (ILP)* of processors.

Analytical performance models are based on mathematical models of microarchitectural components and are usually based on probability distributions. Noonburg and Shen [50], [51] developed an analytical model of superscalar processor performance that is based on Jouppi's idea [37] to consider the dependency between the ILP of programs and the ILP in processors: Typically, the performance (measured by ILP) grows linearly with the number of processor components, but becomes constant when the program's ILP is reached. The performance model of Noonburg and Shen [50], [51] considers probability distributions for branching, instruction fetching and instruction issuing. Dubey et al. [18] focus on the impact of the processor's instruction window size to search for independent instructions, and Zyuban et al. [75] study the impact of data dependency distances in this context. Michaud et al. [47] found that the fetch rate grows approximately as the square root of the distance between mispredicted branches and is proportional to the available ILP in a fixed-size instruction window. The performance model of Karkhanis and Smith [39] focuses additionally on caches, but also on branch predictors, instruction issue width, and reorder buffers. Finally, an even more complete model is presented by Taha and Wills [65] which also considers the number and cycle-accurate behavior of PUs, [10] computes in-order vs. out-of-order superscalarity, and [49] even considers the use of shared memory.

In contrast to analytical performance models, there are *empirical performance models* which aim to reduce the number or length of simulation runs by modeling characteristics such as the basic block size, branch probabilities, and memory access rates. In particular, *statistical simulation* [19], [20], [52], [53], [73] approximates the execution characteristics of a given program by a profile that is used during the design space exploration to generate a shorter synthetic program with the same characteristics. In *sampled program simulation* [27], [28], [61], parts of the simulated program are identified that have the same average characteristics as the entire program, and a weighted average of such program regions is computed as the overall result.

Finally, there are *trend models* that study processor performance by simulating some randomly selected processor instances and extrapolating the observed trends to unknown instances. For the extrapolation, different methods are used such as linear [45] and non-linear regression [36] and also artificial neural networks [35].

In addition to processor performance models, there are also performance models for general parallel computing such as Amdahl's law [1] and Gustafson's law [26]. However, they do not consider architectural parameters of processors, but consider parameters of the programs like the size and structure of basic blocks to study the ILP of general programs [44], [68], [72]. Finally, there are also *performance models for on-chip networks* [4], [46] that become more and more important for manycore processors.

B. Buffered Exposed Datapath (BED) Architectures

Exposed datapath architectures [5], [7], [12], [14]–[16], [22], [24], [32], [38], [55], [58], [66], [67], [71] are relatively new processor architectures that are descendants of VLIW processors [14]–[16]. *They expose their microarchitecture to the compiler so that the compiler is able to allocate processing units (PUs) for the instruction execution, to schedule the instructions on the individual PUs, and to trigger the communication of intermediate results between PUs.* Particular architectures such as TTAs [14]–[16], [29], [32], [38], [77], RAW/Tilera [66], [67], [71], TRIPS [12], [23], [55], [56], Tartan [48], DySER [24], FlexCore [69], AMIDAR [22], STA [13], SCAD [2], [5]–[7], [58], [60], Mozart [57], and others [21], [25], [30], [31], [70] use quite different concepts at a first glance. However, their common idea is to decentralize all processor components to avoid bottlenecks in the hardware design [54], [76], and therefore to use powerful on-chip networks to connect the PUs to make use of a large number of PUs [12], [55]. The compiler not only schedules the instructions, but also allocates the PUs, and handles the communication of intermediate results between PUs. In addition, these architectures are *hybrid dataflow/von Neumann architectures* [8], [11], [34], [74] which means that they still execute sequential programs, but take advantage of ILP through dataflow computing which is enabled by the buffered communication of the PUs that ultimately triggers the computations.

A general template of a *buffered exposed datapath (BED)* architecture is shown in Figure 1: BED architectures consist of a data memory accessed by a load-store unit (LSU), a program memory accessed by a control unit (CU), and a number of general-purpose or special-purpose PUs. All PUs, the LSU, and the CU are connected via FIFO buffers through an interconnection network with each other to avoid an

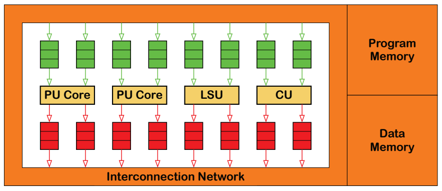

Fig. 1. General Template of a BED architecture [59]

unnecessary synchronization. In many cases, the compiler generates *move code instructions* $src \rightarrow tgt$. The target tgt is thereby an input buffer of one of the PUs, the LSU or the CU, and src may be an output buffer, an opcode or an immediate operand. It is sufficient that PUs have two input buffers `PU[i].inL`, `PU[i].inR` and two output buffers `PU[i].outL`, `PU[i].outR` as well as a special input buffer `PU[i].opc` for the opcodes. PUs fire if they find operands in their input buffers, and produce results that are stored in their output buffers. The interconnection network takes care of the data transport and will move available values from output buffers to input buffers. For example, Figure 2 shows a move code program using four PUs to add the sum of the first eight natural numbers.

While the hardware design of BED processors is simplified compared to other architectures, the compilers are challenged to produce efficient and correct code [58], [59]. Traditional compilers focus mainly on the efficient use of registers which is achieved by depth-first traversals of dataflow graphs. However, this reduces the use of ILP which is optimized by breadth-first traversals of dataflow graphs. In this paper, we consider an optimal code generation that is based on breadth-first traversals of dataflow graphs.

III. A Performance Model for BED Architectures

In this section, we develop a performance model to estimate the runtime of a program on BED machines. The performance model defined in Proposition 1 considers several parameters of a BED machine such as the number of PUs p, the size of each FIFO buffer β, the number of cycles λ required to execute an average instruction, and the maximum number of move instructions w provided by the control unit per cycle.

A. Program Parameters

The number of move instructions n in the program is obviously the most important parameter, but there is also the ILP α of the program which determines how many instructions can be executed in parallel on average. If $\alpha = n$ would hold, then all instructions would be completely independent of each other and the entire program could be executed in a single step using sufficiently many PUs. However, even in this case, the number of PUs can be much smaller than n because many of the move instructions are data transfers that are handled solely by the on-chip network of the BED architecture. Therefore, we distinguish between the total number n of instructions in the program, and the included node firings $f = \mu \cdot n < n$. For a fully parallel execution within a single step, we would therefore only need f PUs since the remaining instructions are handled by the data transfer network. As program parameters, we therefore consider the number of move code instructions n, the fraction of contained firings $\mu = \frac{f}{n}$, and the average ILP α, i.e., the average number of instructions executed in parallel.

B. Architecture Parameter 1: Instruction Latency

The instruction latency λ is the number of cycles it takes to execute an instruction. Typically, the latency depends on the type of instruction, since branches, load/store instructions, and complex arithmetic instructions typically take more time than token moves or simple arithmetic instructions. On the other hand, it does not depend on the other parameters, nor does it depend on a particular program. For the experiments, we assume that load/store instructions take 12 cycles, division takes 8 cycles, multiplication takes 5 cycles, and all other instructions take 3 cycles, which is justified by our hardware prototype. Since the simpler instructions are much more frequent than others, typical values of λ can be close to 3. Therefore, we consider the instruction latency λ as a constant in the following, unless the benchmarks contain many complex operations.

C. Architecture Parameter 2: Number of Processing Units

An obvious parameter of a BED architecture is the number of available PUs p. While the PUs can generally provide specific functionalities, we consider only general-purpose PUs in this paper to better analyze the impact of their number. Due to the design of BED architectures, chip size and power consumption grow only linearly with the number of PUs. Typically, the runtime decreases until the processor reaches the limit of the ILP provided by the program, and then remains constant, so that more PUs do not increase performance [50], [51]. Therefore, for a fixed program, we investigate how the runtime depends on the number of PUs. In particular, we want to find out whether the BED machine is able to use the entire ILP of the program, and how the performance grows with limited resources.

In the best case, all instructions are independent of each other and the runtime of a program with n instructions can be computed in time $t \approx \frac{n \cdot \lambda}{p}$. However, according to Amdahl's law, this is typically unrealistic, since there are usually some sequential steps in programs. If the ILP allows on average only α of the n instructions to be executed at a time, then the runtime will instead be $t \approx \frac{n}{\alpha}$. Furthermore, the PUs are only responsible for firing nodes, and among the α instructions that can be executed per cycle, there are only $\mu \cdot \alpha$ many node firings that can be handled by the PUs. For this reason, we get the following estimation for a fully utilized system

$$\frac{p}{\lambda} \approx \mu \cdot \alpha$$

Based on this, we obtain the following estimation for the runtime that we will refine further in the following:

$$t \geq \frac{\mu \cdot n}{\min(\mu \cdot \alpha, \frac{p}{\lambda})} = \max\left\{\frac{n}{\alpha}, \frac{\mu \cdot \lambda \cdot n}{p}\right\}$$

D. Architecture Parameter 3: Instruction Issue Width

The instruction issue width $\hat{w}$ is the maximum number of move instructions that will be fetched and sent to the PUs in one processor cycle. The actual number of instructions issued may be less than the maximum because the buffers of the PUs may be full. An obvious, but typically unrealistic, upper bound for the issue width is the size of the program. In this case, all move instructions are fetched and sent to the buffers of the PUs in the first cycle, and the rest of the execution can then be completely organized along the availability of operands, i.e., in dataflow order, to achieve maximum ILP.

In a balanced system, the number of instructions w provided on average by the control unit is equal to the number of instructions consumed by the PUs. Since p PUs with an average instruction latency λ can execute on average $\frac{p}{\lambda}$ instructions, but only $\mu \cdot w$ of the provided instructions are real node firings, we find the following relationship

$$\frac{p}{\lambda} \approx \mu \cdot w \approx \mu \cdot \alpha$$

Hence, if we provide only $w < \frac{p}{\mu \cdot \lambda}$ instructions per cycle, some of the PUs will become idle, and we would achieve the same performance with fewer PUs as well.

E. Architecture Parameter 4: Buffer Size

As already mentioned in the previous section, the maximum buffer size $\hat{\beta}$ is another parameter of a BED architecture. It directly affects the issue width and the possible parallelism of the BED machine. In particular, the buffers must be large enough to store the instructions that are issued in each cycle so that with two buffers per PU, we need the following with the average number of tokens β in the buffers:

$$2 \cdot \beta \cdot p \approx w$$

F. Performance Model

Having discussed the parameters of the BED architecture and the programs executed on it, we now summarize the previous discussions in a performance model: We consider the execution of a program with n instructions containing $\mu \cdot n$ node firings and an average ILP α as determined by data dependencies. The program is executed on a BED architecture with p PUs, instruction issue width w, instruction latency λ, and buffer size β. Since we are considering a particular program and want to determine an optimal BED machine for its execution, we consider n, μ, and α to be constants. The instruction latency λ is also considered to be a constant, since it depends only on the circuit implementation of the arithmetic operations. In addition to the average values, we also want to determine the minimum number p of PUs, the minimum instruction issue width w, and the minimum buffer size β such that the minimum runtime t is obtained for the program under consideration.

For a processor architecture with unlimited resources, the program can be executed in time $t = \frac{n}{\alpha}$, so the real runtime on a BED machine must be greater than this value. For limited resources, the following potential bottlenecks exist:

1) The number of PUs may not be sufficient to perform $\mu \cdot \alpha$ node firings per cycle. Since the BED machine is able to execute $\frac{p}{\lambda}$ node firings per cycle, we should have $\frac{p}{\lambda} \approx \mu \cdot \alpha$.
2) The control unit must provide enough instructions per cycle, thus $w \approx \alpha$.
3) The size of the buffers β is not sufficient to store w instructions per cycle, thus, we demand $2 \cdot \beta \cdot p \approx w$.

We therefore obtain the following performance model:

Proposition 1 (Performance Model)**.** *The runtime t of a program with n instructions, $\mu \cdot n$ node firings, and average ILP α on a BED machine with p PUs, buffer size β, instruction latency λ, and instruction issue width w is determined as follows:*

$$t(p, w, \beta) = \frac{n}{\min\{\alpha, \frac{p}{\mu \cdot \lambda}, w, 2p\beta\}}$$

BED machines should therefore be designed with parameters p,w,β,λ such that the following equations hold:

$$\frac{p}{\mu \cdot \lambda} = w = 2 \cdot \beta \cdot p$$

The perfect BED machine will therefore use $p = \alpha \cdot \mu \cdot \lambda$ PUs with an instruction issue width $w := \alpha = \frac{p}{\mu \cdot \lambda}$ and buffer size $\beta := \frac{\alpha}{2p} = \frac{w}{2p} = \frac{1}{2\mu\lambda}$. With these estimations, we can already see that the buffer size β is quite uncritical, i.e., it can be chosen independent of p and w, and depends only on μ and λ. In contrast, the instruction issue width w must grow linearly with the number of PUs p.

In the following, we want to validate the above equations with experiments. As mentioned above, we therefore consider the execution of a program with n instructions, $\mu \cdot n$ node firings, and average ILP α. For the execution, we can therefore consider two phases [37], [50], [51]:

a) Full Utilization: is obtained for parameters $\min\{\frac{p}{\mu \cdot \lambda}, w, 2p\beta\} \leq \alpha$. For these parameter values, all p PUs are fully utilized, thus the BED machine executes $\frac{p}{\mu \cdot \lambda} \leq \alpha$ instructions per cycle, the control unit issues $w \leq \alpha$ instructions per cycle, and the FIFO buffers store these instructions per cycle. For the runtime, we have

$$t(p, w, \beta) = \frac{n}{\min\{\frac{p}{\mu \cdot \lambda}, w, 2p\beta\}}$$

Thus, the runtime depends anti-proportionally on p and w.

b) Underutilized Execution: is obtained for parameters $\min\{\frac{p}{\mu \cdot \lambda}, w, 2p\beta\} > \alpha$ where the BED machine could execute more instructions per cycle than the program allows. Hence, the runtime converges to its lower bound $t = \frac{n}{\alpha}$. Thus, for a given program with a high degree of average ILP α, we will see that the runtime $\mu \cdot \lambda \cdot \frac{n}{p}$ decreases anti-proportionally with a growing number of PUs p until we reach full utilization. After that point, more PUs will not improve the performance. Also, if the instruction width is less than α, it will limit the performance that reaches its lower bound $\frac{n}{w} > \frac{n}{\alpha}$.

```
1 -> PU[0].inL
2 -> PU[0].inR
AddN -> PU[0].opc
3 -> PU[1].inL
4 -> PU[1].inR
AddN -> PU[1].opc
5 -> PU[2].inL
6 -> PU[2].inR
AddN -> PU[2].opc
7 -> PU[3].inL
8 -> PU[3].inR
AddN -> PU[3].opc
PU[0].outL -> PU[0].inL
PU[1].outL -> PU[0].inR
AddN -> PU[0].opc
PU[2].outL -> PU[1].inL
PU[3].outL -> PU[1].inR
AddN -> PU[1].opc
PU[0].outL -> PU[0].inL
PU[1].outL -> PU[0].inR
AddN -> PU[0].opc
```

Fig. 2. Move Code for Adding Numbers 1,...,8 using p=4 PUs in TreeSum.

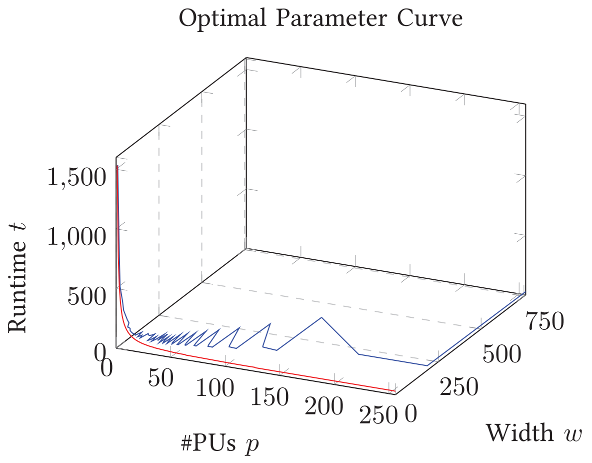

Fig. 3. Optimal Parameter Curve for the Tree Summation of 512 Numbers (blue) and its projection to the runtime-PU plane (red).

IV. Experimental Results

In this section, we consider three benchmarks to validate our performance model stated in Proposition 1. The programs are given as dataflow graphs that are leveled according to their data dependencies. These dataflow graphs can be mapped to a BED machine with a single PU, and for a given number of PUs p, the nodes are scheduled level by level by allocating one of the available PUs (list scheduling is known to be optimal [33], [42], [64] in this case). We then determine for triples (p, w, β) the execution time $t(p, w, \beta)$ with a cycle-accurate instruction set simulator. In all experiments, the values for the buffer size β turned out to be less important, so that we focus on a mapping of (p, w) to the runtime t. Instead of presenting 3D-surface maps, we prefer to extract the optimal 3D-curve of points (p, w, t) consisting for a given p the minimum width w to achieve the minimum runtime t.

A. Benchmark 1: Tree Sum

As a first benchmark, we consider the summation of $m = 2^k$ numbers organized in a binary tree of height k. The move code program requires $m - 1$ additions and $2(m - 1)$ move instructions for transferring the intermediate results (see Figure 2). In general, there are $n = 3(m - 1)$ instructions containing $m - 1$ additions and $2(m - 1)$ data transfers, so that we have $\mu = \frac{m-1}{3(m-1)} = \frac{1}{3}$. If we would execute all nodes of a level of the binary tree in parallel, we would obtain the minimum runtime $t_{min} = k \cdot \lambda$. Since all $n = 3(m - 1) = 3(2^k - 1)$ instructions are executed in this time, we obtain $\alpha = \frac{n}{t_{min}} = \frac{3(2^k-1)}{k}$.

For example, for $k = 9$, we add $m = 2^9 = 512$ numbers organized in a binary tree with $m - 1 = 511$ addition nodes that are arranged in $k = 9$ levels. The minimum runtime of the move code program with $n = 3(m - 1) = 1533$ instructions is $t_{min} = k \cdot \lambda = 9 \cdot 3 = 27$ and the average ILP is $\alpha = \frac{n}{t_{min}} = \frac{1533}{27} \approx 56.77\ldots$.

Using our simulator, we compute triples (p, w, t) consisting of the runtime t obtained for numbers of PUs p and instruction issue widths w which yields a surface $t(p, w)$. For the obtained triples (p, w, t) on that surface, we compute for each p the minimum w to obtain the minimum runtime t which yields a curve in the coordinates (p, w, t) that is shown in Figure 3. The points on this curve determine for any number of PUs p, the minimum width w at which the minimum runtime t was obtained. For this benchmark, the minimum instruction width w has the upper bound $3p$. In between the values p with $w = 3p$, we see instruction issue widths $w < 3p$. They result from the possibility to distribute the nodes on more PUs. Thus, some PUs can start their executions later without increasing the runtime. Since the minimal w strongly depends on the number of PUs starting to execute in the first cycle, this reduces the required w.

As expected, the minimum runtime $t = 27$ is reached for $p \geq 256$ with $w \geq 768$. However, for $128 \leq p < 256$, we already find a slightly higher runtime $t = 30$. Hence, with almost half the number of PUs and instruction width, we get almost the same performance! Also for $64 \leq p \leq 127$, we get acceptable runtimes $39 \geq t \geq 33$. So, everything is in accordance with our performance model: Considering Proposition 1, we expect a linear dependency between p and w which is confirmed by the experiment.

B. Benchmark 2: Parallel Prefix Computation

The parallel prefix computation is a fundamental algorithm of parallel computing [41], [43], [63]: The task is to compute, for a given sequence of operands $x_0, \ldots, x_{n-1}$ and an associative function f, the sequence of prefixes x_0, $f(x_0, x_1)$, $f(x_0, f(x_1, x_2))$, ..., $f(x_0, f(\ldots, x_{n-1})\ldots)$. With sufficiently many PUs, it is possible to solve the problem in $O(\log(n))$ time, and many algorithms of this complexity have been proposed, including those by Sklansky [62], Kogge-Stone [40], Brent-Kung [9], and Ladner-Fisher [43].

In this section, we evaluate the Brent-Kung and Kogge-Stone algorithms represented as dataflow graphs in Figure 4 and Figure 5, respectively, on a BED machine where we use a

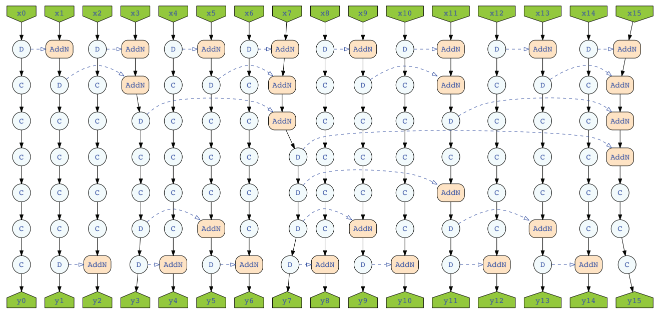
x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15
D
C
AddN
y0 y1 y2 y3 y4 y5 y6 y7 y8 y9 y10 y11 y12 y13 y14 y15

Fig. 4. Parallel Prefix Computation due to Brent-Kung for 16 Inputs.

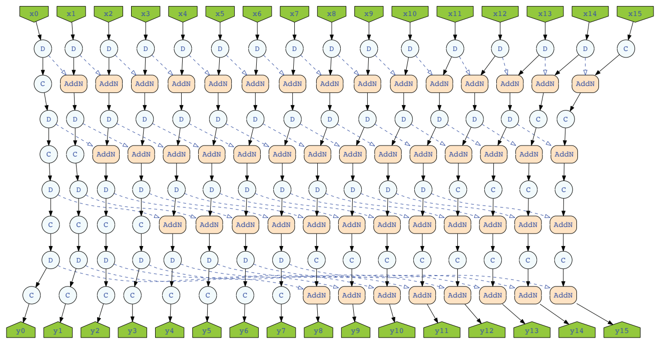
x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15
D
C
AddN
y0 y1 y2 y3 y4 y5 y6 y7 y8 y9 y10 y11 y12 y13 y14 y15

Fig. 5. Parallel Prefix Computation due to Kogge-Stone for 16 Inputs.

simple addition for the function f. The dataflow graph of the Brent-Kung algorithm as shown in Figure 4 has $2 \cdot \log_2(n) - 1$ rows which have however dependencies from left to right as shown in the graph. Hence, we add further copy nodes to make sure that the dependencies are not within a row so that each row can be fired in parallel. After this transformation, there are $n \cdot (4 \cdot \log_2(n) - 3)$ nodes in total including $2 \cdot n - \log_2(n) - 2$ addition nodes, the same number of duplication nodes, and $(4 \cdot n + 2) \cdot \log_2(n) - 7 \cdot n + 4$ copy nodes. The longest path through the dataflow graph starts in x_0 and leads to y_{n-2} with $4(\log_2(n) - 1) + 1$ many nodes, so that the minimum runtime is $t = (4 \cdot \log_2(n) - 3) \cdot \lambda$. Since we need two move instructions for copy and duplication nodes, and three move instructions for addition nodes, the program has $(8 \cdot n - 1) \cdot \log_2(n) - 4 \cdot n - 2$ move instructions. The maximum ILP is therefore $\alpha = \frac{(8 \cdot n-1) \cdot \log_2(n) - 4 \cdot n - 2}{(4 \log_2(n) - 3) \cdot \lambda} \in \Theta(\frac{n}{\lambda})$, and $\mu = \frac{n \cdot (4 \cdot \log_2(n) - 3)}{(8 \cdot n - 1) \cdot \log_2(n) - 4 \cdot n - 2} \approx \frac{1}{2}$.

The dataflow graph of the Kogge-Stone algorithm has $2 \cdot \log_2(n)$ levels with $n \cdot (\log_2(n) - 1) + 1$ addition nodes, the same number of duplication nodes, and $2 \cdot n - 2$ copy nodes, thus $2 \cdot n \cdot \log_2(n)$ nodes in total. Since there are $5 \cdot n \cdot \log_2(n) - n + 1$ instructions, we get $\mu = \frac{2 \cdot n \cdot \log_2(n)}{5 \cdot n \cdot \log_2(n) - n + 1}$. The length of the longest path through the dataflow graph has $2 \cdot \log_2(n)$ nodes which determines the minimum runtime $t = 2 \cdot \lambda \cdot \log_2(n)$.

TABLE I
STATISTICS FOR BRENT-KUNG AND KOGGE-STONE FOR 128 NUMBERS

	Brent-Kung	Kogge-Stone
nodes f	3200	1792
rows	13	14
depth δ	25	14
min. runtime $t = \delta \cdot \lambda$	75	42
instructions n	6647	4353
$\mu = \frac{f}{n}$	0.48	0.41
ILP α	88.6	103.6

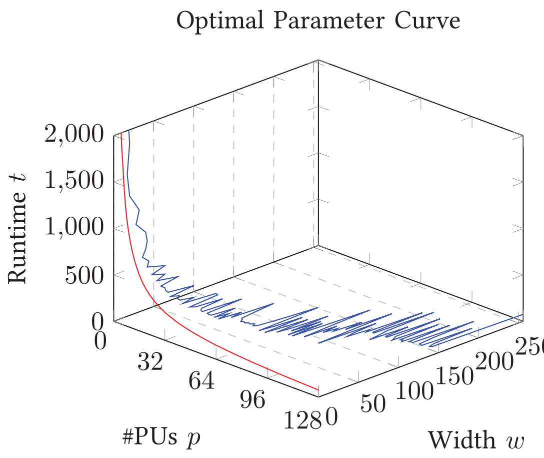

Fig. 6. Optimal Parameter Curve for the Brent-Kung Parallel Prefix Sum of 128 Numbers (blue) and its projection to the runtime-PU plane (red).

Table I shows some statistics for both dataflow graphs for 128 inputs. Since both graphs have 128 nodes in each level, 128 PUs are required for the minimum execution time.

The optimal parameter curves for 128 numbers are shown in Figures 6 and 7, respectively, and are similar to the curve of the binary tree sum. Their projections onto the PU-time dimensions are shown in Figure 8, where the blue curve is obtained for Brent-Kung and the red curve for

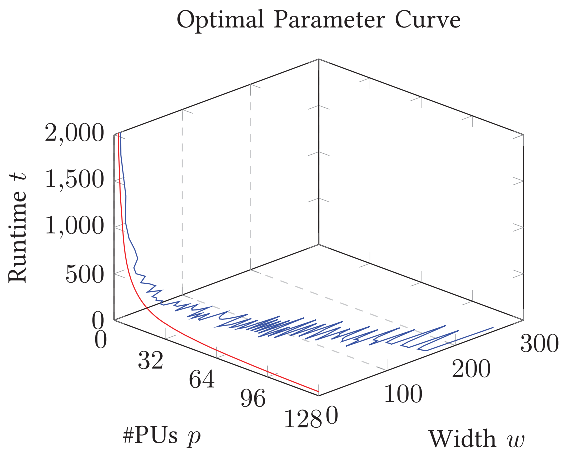

Fig. 7. Optimal Parameter Curve for the Kogge-Stone Parallel Prefix Sum of 128 Numbers (blue) and its projection to the runtime-PU plane (red).

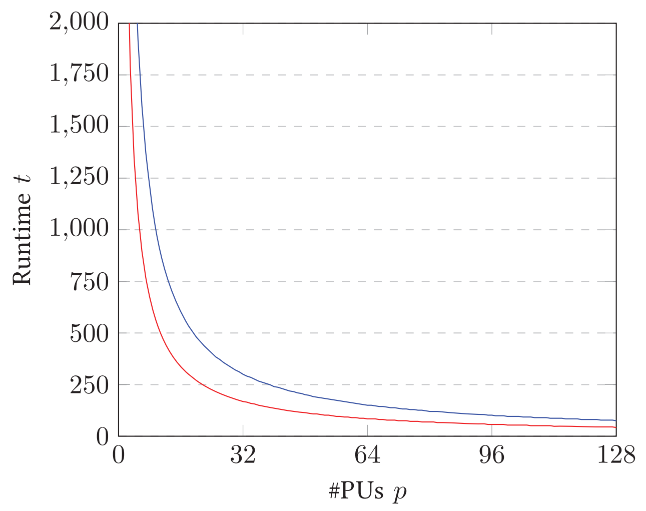

Fig. 8. Optimal Runtime in Terms of the PU Number for the Parallel Prefix Sum of 128 Numbers (blue: Brent-Kung; red: Kogge-Stone).

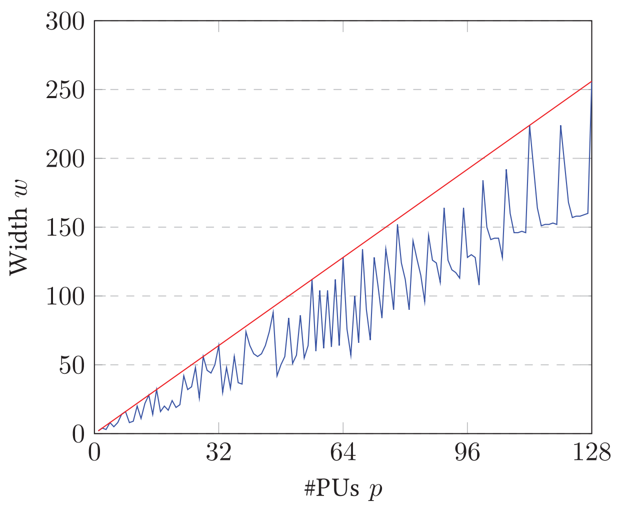

Fig. 10. Instruction Issue Width in Terms of PU Number for the Kogge-Stone Parallel Prefix Sum of 128 Numbers.

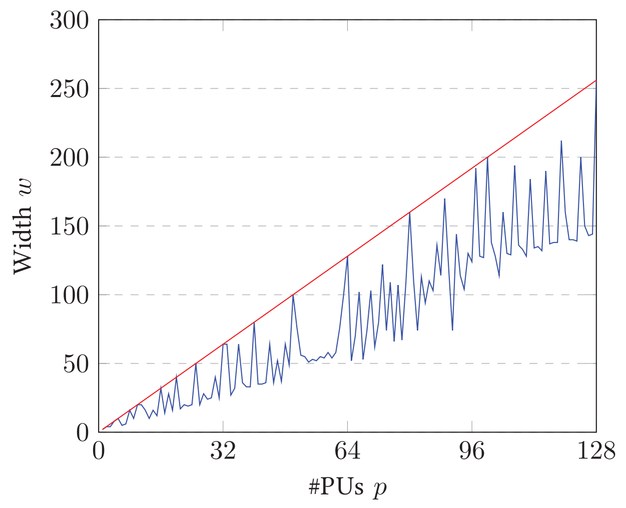

Fig. 9. Instruction Issue Width in Terms of PU Number for the Brent-Kung Parallel Prefix Sum of 128 Numbers.

Kogge-Stone. The two curves show the anti-proportional dependency between the runtime and the number of PUs as predicted by our performance model, where the minimum runtimes obtained for $p \geq 128$ are 75 cycles (Brent-Kung) and 42 cycles (Kogge-Stone), respectively.

Figures 9 and 10 show the projections of the optimal parameter curves shown in Figures 6 and 7 onto the PU-width dimensions. As can be seen, we can find for each benchmark a constant κ such that $w(p) \leq \kappa \cdot p$ which confirms the linear dependency between the width w and the number of PUs p as predicted by our model.

C. Benchmark 3: Odd-Even Sorting

Odd-even sorting is a relatively simple parallel sorting algorithm that requires $O(n^2)$ work with a parallel runtime $O(n)$ using $O(n)$ PUs. The algorithm performs a sequence of compare&swap operations in pairs of rounds where in the first round of such a pair all elements with even/odd indices `x[2*i]`,`x[2*i+1]` are compared, and in the second round of the pair, all elements with odd/even indices `x[2*i+1]`,`x[2*i+2]` are compared. Thus, odd-even sorting of n numbers requires $\lfloor \frac{n}{2} \rfloor$ and $\lfloor \frac{n-1}{2} \rfloor$ compare&swap operations in each first and second round, respectively, (see Figure 11). The dataflow graph has $\frac{1}{2} \cdot (27 \cdot n^2 - 13 \cdot n)$ nodes and its move code program has $\frac{1}{2} \cdot (61 \cdot n^2 - 33 \cdot n)$ instructions. Hence, $\mu \approx 0.44$. Moreover, we have n levels of compare&swap subgraphs and each subgraph has 7 levels. Thus, we have $7n$ levels, and the minimum runtime is $7\lambda n$.

Figures 12 and 13 show the results for sorting 16 numbers. The dataflow graph has 3352 nodes with at most $8 \cdot 6 = 48$ nodes in a level, and its move code program has 7544 instructions. The curves show a precise anti-proportional dependency between the minimum runtime t and the number of PUs p as predicted by our performance model. The minimum runtime $t = 7 \cdot 16 \cdot 3 = 336$ cycles is achieved with $p \geq 48$ PUs.

Clearly, BED architectures have a fixed number of PUs p and a fixed instruction issue width w. According to our performance model, p and w determine each other in a balanced BED architecture. Otherwise, either more instructions than necessary are issued, or not all PUs can be utilized. This phenomenon is shown in Figure 13 where the runtime depending on the number of PUs is shown for fixed instruction issue widths $w \in \{4, 8, 16, 32\}$ given in red, blue, brown, and green color. These curves follow the optimal runtime until a larger width is required, and then remains constant because too few instructions are issued to utilize further PUs. While this should be clear, it is noteworthy that we get the optimal runtime $t = 336$ with $p \geq 48$ PUs only for $w \geq 32$. This instruction width is not reasonable because of the anti-proporitional dependency between t and w that gives already good runtimes for much smaller widths.

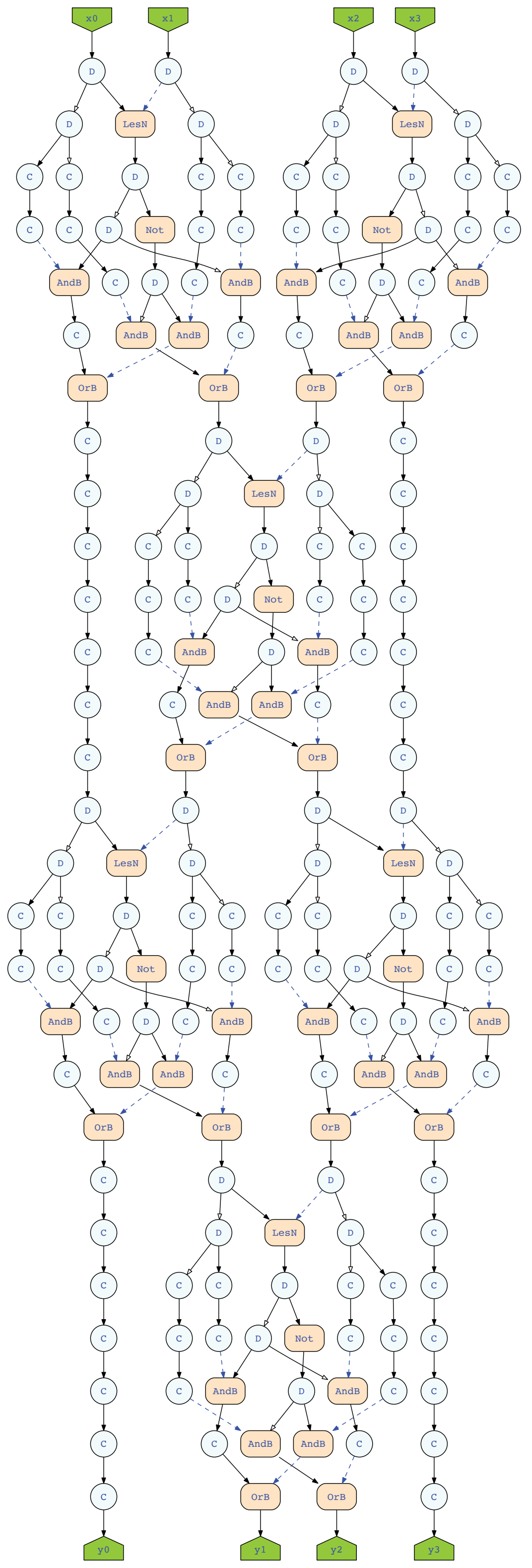
Fig. 11. Dataflow Graph for the OddEven-Sorting with 4 Inputs.

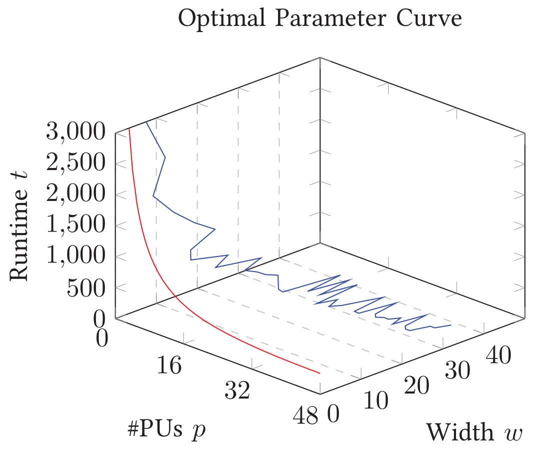

Fig. 12. Optimal Parameter Curve for Odd-Even-Sorting of 16 Numbers (blue) and its projection to the runtime-PU plane (red).

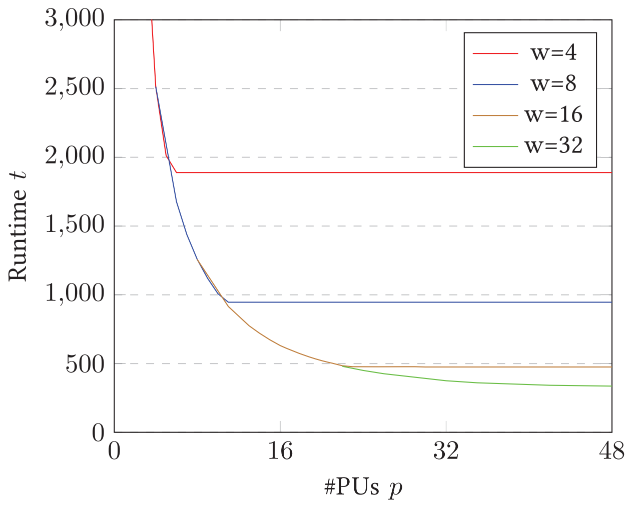

Fig. 13. Runtime in Terms of PU Number for fixed Instruction Issue Widths for the Odd-Even-Sorting of 16 Numbers.

V. Conclusions

This paper presents a performance model for buffered exposed datapath (BED) processors. The model relates relevant design parameters such as the number of PUs, the instruction issue width, the size of FIFO buffers, the instruction latency, the computation/communication ratio with the execution time and the ILP of programs. Our benchmarks perfectly validated the proposed relationships.

The performance model also shows that BED architectures can utilize a large number of PUs to take full advantage of programs' ILP, and they require an instruction issue width that grows proportionally to the number of PUs. Second, there is a anti-proportional dependency between the runtime and the number of PUs. Therefore, it is not reasonable to determine design parameters for a minimum runtime since a comparable runtime can often be achieved with much smaller parameter values.

References

[1] Amdahl, G. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Joint Computer Conferences* (1967), ACM, pp. 483–485.

[2] Anders, M., Bhagyanath, A., and Schneider, K. On memory optimal code generation for exposed datapath architectures with buffered processing units. In *Application of Concurrency to System Design (ACSD)* (Bratislava, Slovakia, 2018), T. Chatain and R. Grosu, Eds., IEEE Computer Society, pp. 115–124.

[3] Austin, T., Larson, E., and Ernst, D. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer 35*, 2 (2002), 59–67.

[4] Banerjee, N., Vellanki, P., and Chatha, K. A power and performance model for Network-on-Chip architectures. In *Design, Automation and Test in Europe (DATE)* (Paris, France, 2004), IEEE Computer Society, pp. 1250–1255.

[5] Bhagyanath, A., and Schneider, K. Optimal compilation for exposed datapath architectures with buffered processing units by SAT solvers. In *Formal Methods and Models for Codesign (MEMOCODE)* (Kanpur, India, 2016), E. Leonard and K. Schneider, Eds., IEEE Computer Society, pp. 143–152.

[6] Bhagyanath, A., and Schneider, K. Exploring different execution paradigms in exposed datapath architectures with buffered processing units. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (Pythagorion, Greece, 2017), Y. Patt and S. Nandy, Eds., IEEE Computer Society, pp. 1–10.

[7] Bhagyanath, A., and Schneider, K. Exploring the potential of instruction-level parallelism of exposed datapath architectures with buffered processing units. In *Application of Concurrency to System Design (ACSD)* (Zaragoza, Spain, 2017), A. Legay and K. Schneider, Eds., IEEE Computer Society, pp. 106–115.

[8] Blake, G., Dreslinski, R., and Mudge, T. A survey of multicore processors. *IEEE Signal Processing Magazine 26*, 6 (November 2009), 26–37.

[9] Brent, R., and Kung, H. A regular layout for parallel adders. *IEEE Transactions on Computers (T-C) 31*, 3 (1982), 260–264.

[10] Breughe, M., Eyerman, S., and Eeckhout, L. A mechanistic performance model for superscalar in-order processors. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)* (New Brunswick, NJ, USA, 2012), IEEE Computer Society, pp. 14–24.

[11] Buehrer, R., and Ekanadham, K. Incorporating dataflow ideas into von Neumann processors for parallel execution. *IEEE Transactions on Computers (T-C) 36*, 12 (December 1987), 1515–1522.

[12] Burger, D., Keckler, S., McKinley, K., Dahlin, M., John, L., Lin, C., Moore, C., Burrill, J., McDonald, R., and Yoder, W. Scaling to the end of silicon with EDGE architectures. *IEEE Computer 37*, 7 (July 2004), 44–55.

[13] Cichon, G., Robelly, P., Seidel, H., Matús, E., Bronzel, M., and Fettweis, G. Synchronous transfer architecture (STA). In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (Samos, Greece, 2004), A. Pimentel and S. Vassiliadis, Eds., vol. 3133 of *LNCS*, Springer, pp. 343–352.

[14] Corporaal, H. Design of transport triggered architectures. In *Great Lakes Symposium on VLSI (GLSVLSI)* (Notre Dame, IN, USA, 1994), IEEE Computer Society, pp. 130–135.

[15] Corporaal, H. TTAs: Missing the ILP complexity wall. *Journal of Systems Architecture 45*, 12-13 (June 1999), 949–973.

[16] Corporaal, H., Janssen, J., and Arnold, M. Computation in the context of transport triggered architectures. *International Journal of Parallel Programming 28*, 4 (August 2000), 401–427.

[17] Degawa, Y., Koizumi, T., Nakamura, T., Shioya, R., Kadomoto, J., Irie, H., and Sakai, S. Accurate and fast performance modeling of processors with decoupled front-end. In *International Conference on Computer Design (ICCD)* (Storrs, CT, USA, 2021), IEEE Computer Society, pp. 88–92.

[18] Dubey, P., Adams, G., and Flynn, M. Instruction window size tradeoffs and characterization of program parallelism. *IEEE Transactions on Computers 43*, 4 (April 1994), 431–442.

[19] Eeckhout, L., Bell, R., Stougie, B., De Bosschere, K., and Kurian John, L. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *International Symposium on Computer Architecture (ISCA)* (Munich, Germany, 2004), IEEE Computer Society, pp. 350–363.

[20] Eeckhout, L., Nussbaum, S., Smith, J., and de Bosschere, K. Statistical simulation: adding efficiency to the computer designer's toolbox. *IEEE Micro 23*, 5 (2003), 26–38.

[21] Emami, M., Bezati, E., Janneck, J., and Larus, J. Triggered scheduling: Efficient detection of dataflow network idleness on heterogeneous systems. In *Field-Programmable Gate Arrays (FPGA)* (Virtual Event, 2021), L. Shannon and M. Adler, Eds., ACM, pp. 226–227.

[22] Gatzka, S., and Hochberger, C. The AMIDAR class of reconfigurable processors. *The Journal of Supercomputing 32*, 2 (2005), 163–181.

[23] Gebhart, M., Maher, B., Coons, K., Diamond, J., Gratz, P., Marino, M., Ranganathan, N., Robatmili, B., Smith, A., Burrill, J., Keckler, S., Burger, D., and McKinley, K. An evaluation of the TRIPS computer system. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Washington, District of Columbia, USA, 2009), M. Soffa and M. Irwin, Eds., ACM, pp. 1–12.

[24] Govindaraju, V., Ho, C.-H., Nowatzki, T., Chhugani, J., Satish, N., Sankaralingam, K., and Kim, C. DySER: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro 33*, 5 (2012), 38–51.

[25] Govindaraju, V., Ho, C.-H., and Sankaralingam, K. Dynamically specialized datapaths for energy efficient computing. In *International Symposium on High Performance Computer Architecture* (San Antonio, TX, USA, 2011), IEEE Computer Society, pp. 503–514.

[26] Gustafson, J. Reevaluating Amdahl's law. *Communications of the ACM (CACM) 31*, 5 (May 1988), 532–533.

[27] Hamerly, G., Perelman, E., Lau, J., and Calder, B. SimPoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction-Level Parallelism 7* (2005), 1–28.

[28] Hamerly, G., Perelman, E., Lau, J., Calder, B., and Sherwood, T. Using machine learning to guide architecture simulation. *Journal of Machine Learning Research 7*, 12 (2006), 343–378.

[29] He, Y., She, D., Mesman, B., and Corporaal, H. MOVE-pro: A low power and high code density TTA architecture. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (Samos, Greece, 2011), L. Carro and A. Pimentel, Eds., IEEE Computer Society, pp. 294–301.

[30] Hepola, K., Multanen, J., and Jääskeläinen, P. Dual-IS: Instruction set modality for efficient instruction level parallelism. In *Architecture of Computing Systems (ARCS)* (Heilbronn, Germany, 2022), M. Schulz, C. Trinitis, N. Papadopoulou, and T. Pionteck, Eds., vol. 13642 of *LNCS*, Springer, p. 17–32.

[31] Hepola, K., Multanen, J., and Jääskeläinen, P. Energy-efficient exposed datapath architecture with a RISC-V instruction set mode. *IEEE Transactions on Computers 73*, 2 (February 2024), 560–573.

[32] Hoogerbrugge, J., and Corporaal, H. Transport-triggering vs. operation-triggering. In *Compiler Construction (CC)* (Edinburgh, UK, 1994), P. Fritzson, Ed., vol. 786 of *LNCS*, Springer, pp. 435–449.

[33] Hu, T. Parallel sequencing and assembly line problems. *Operations Research 9*, 6 (1961), 841–848.

[34] Iannucci, R. Towards a dataflow/von Neumann hybrid architecture. In *International Symposium on Computer Architecture (ISCA)* (Honolulu, Hawaii, USA, 1988), H. Siegel, Ed., IEEE Computer Society, pp. 131–140.

[35] Ipek, E., McKee, S., de Supinski, B., and Caruana, R. Efficiently exploring architectural design spaces via predictive modeling. *ACM SIGOPS Operating Systems Review 40*, 5 (2006), 195–206.

[36] Joseph, P., Vaswani, K., and Thazhuthaveetil, M. A predictive performance model for superscalar processors. In *Microarchitecture (MICRO)* (Orlando, Florida, USA, 2006), IEEE Computer Society, pp. 161–170.

[37] Jouppi, N. The nonuniform distriubtion of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers 38*, 12 (December 1989), 1645–1658.

[38] Jääskeläinen, P., Tervo, A., Vayá, G., Viitanen, T., Behmann, N., and Blume, H. Transport-triggered soft cores. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (Vancouver, BC, Canada, 2018), IEEE Computer Society, pp. 83–90.

[39] Karkhanis, T., and Smith, J. A first-order superscalar processor model. In *International Symposium on Computer Architecture (ISCA)* (Munich, Germany, 2004), IEEE Computer Society, pp. 338–349.

[40] Kogge, P., and Stone, H. A parallel algorithm for the efficient solution of a general class of recurrences. *IEEE Transactions on Computers (T-C) 22* (1973), 786–793.

[41] Kruskal, C., Rudolph, L., and Snir, M. The power of parallel prefix. *IEEE Transactions on Computers 34*, 10 (1985), 965–968.

[42] Kwok, Y.-K., and Ahmad, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR) 31*, 4 (December 1999), 406–471.

[43] Ladner, R., and Fischer, M. Parallel prefix computation. *Journal of the ACM (JACM) 27*, 4 (October 1980), 831–838.

[44] Lam, M., and Wilson, R. Limits of control flow on parallelism. In *International Symposium on Computer Architecture (ISCA)* (Gold Coast, Queensland, Australia, 1992), ACM, pp. 46–57.

[45] Lee, B., and Brooks, D. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, California, USA, 2006), J. Shen and M. Martonosi, Eds., ACM, pp. 185–194.

[46] Mandal, S., Ayoub, R., Kishinevsky, M., and Ogras, U. Analytical performance models for NoCs with multiple priority traffic classes. *ACM Transactions on Embedded Computing Systems (TECS) 18*, 5 (October 2019), 52:1–52:21.

[47] Michaud, P., Seznec, A., and Jourdan, S. Exploring instruction fetch bandwidth requirement in wide issue superscalar processors. In *Parallel Architectures and Compilation Techniques (PACT)* (Newport Beach, California, USA, 1999), IEEE Computer Society, pp. 2–10.

[48] Mishra, M., Callahan, T., Chelcea, T., Venkataramani, G., Budiu, M., and Goldstein, S. Tartan: evaluating spatial computation for whole program execution. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, California, USA, 2006), J. Shen and M. Martonosi, Eds., ACM, pp. 163–174.

[49] Mitra, R., Joshi, B., Ravindran, A., Mukherjee, A., and Adams, R. Performance modeling of shared memory multiple issue multicore machines. In *International Conference on Parallel Processing Workshops (ICPP-WS)* (Pittsburgh, Pennsylvania, USA, 2012), IEEE Computer Society, pp. 464–473.

[50] Noonburg, D., and Shen, J. Theoretical modeling of superscalar processor performance. In *Microarchitecture (MICRO)* (San Jose, California, USA, 1994), IEEE Computer Society, pp. 52–62.

[51] Noonburg, D., and Shen, J. A framework for statistical modeling of superscalar processor performance. In *International Symposium on High-Performance Computer Architecture (HPCA)* (San Antonio, TX, USA, 1997), IEEE Computer Society, pp. 298–309.

[52] Nussbaum, S., and Smith, J. Modeling superscalar processors via statistical simulation. In *Parallel Architectures and Compilation Techniques (PACT)* (Barcelona, Catalunya, Spain, 2001), IEEE Computer Society, pp. 15–24.

[53] Oskin, M., Chong, F., and Farrens, M. HLS: combining statistical and symbolic simulation to guide microprocessor designs. In *International Symposium on Computer Architecture (ISCA)* (Vancouver, British Columbia, Canada, 2000), ACM, pp. 71–82.

[54] Rixner, S., Dally, W., Kapasi, U., Mattson, P., and Owens, J. Memory access scheduling. In *International Symposium on Computer Architecture (ISCA)* (Vancouver, British Columbia, Canada, 2000), ACM, pp. 128–138.

[55] Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Ranganathan, N., Burger, D., Keckler, S., Mcdonald, R., and Moore, C. TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. *ACM Transactions on Architecture and Code Optimization (TACO) 1*, 1 (2004), 62–93.

[56] Sankaralingam, K., Nagarajan, R., McDonald, R., Desikan, R., Drolia, S., Govindan, M., Gratz, P., Gulati, D., Hanson, H., Kim, C., Liu, H., Ranganathan, N., Sethumadhavan, S., Sharif, S., Shivakumar, P., Keckler, S., and Burger, D. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Microarchitecture (MICRO)* (Orlando, Florida, USA, 2006), IEEE Computer Society, pp. 480–491.

[57] Sankaralingam, K., Nowatzki, T., Gangadhar, V., Shah, P., Davies, M., Galliher, W., Guo, Z., Khare, J., Vijay, D., Palamuttam, P., Punde, M., Tan, A., Thiruvengadam, V., Wang, R., and Xu, S. The mozart reuse exposed dataflow processor for AI and beyond: industrial product. In *International Symposium on Computer Architecture (ISCA)* (New York, New York, USA, 2022), V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds., ACM, pp. 978–992.

[58] Schneider, K., and Bhagyanath, A. Consistency constraints for mapping dataflow graphs to hybrid dataflow/von Neumann architectures. *Transactions on Embedded Computing Systems (TECS) 22*, 5 (2023), 81:1–81:25.

[59] Schneider, K., Bhagyanath, A., and Roob, J. Code generation criteria for buffered exposed datapath architectures from dataflow graphs. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)* (San Diego, CA, USA, 2022), T. Grosser and K. Lee, Eds., ACM, pp. 133–145. 10.1145/3519941.3535076.

[60] Schneider, K., Bhagyanath, A., and Roob, J. Virtual buffers for exposed datapath architectures. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)* (Virtual Event, 2022), J. Brandt, Ed., vol. 302 of *ITG-Fachbericht*, VDE, pp. 45–55.

[61] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. Automatically characterizing large scale program behavior. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, California, USA, 2002), K. Gharachorloo, Ed., ACM, pp. 45–57.

[62] Sklansky, J. Conditional-sum addition logic. *IRE Transactions on Electronic Computers EC-9* (1960), 226–231.

[63] Snir, M. Depth-size trade-offs for parallel prefix computation. *Journal of Algorithms 7*, 2 (June 1986), 185–201.

[64] Stadtherr, H. *Work Efficient Parallel Scheduling Algorithms*. PhD thesis, Department of Computer Science, TU Munich, March 1998. PhD.

[65] Taha, T., and Wills, S. An instruction throughput model of superscalar processors. *IEEE Transactions on Computers 57*, 3 (2008), 389–403.

[66] Taylor, M. Design decisions in the implementation of a RAW architecture workstation. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, USA, September 1999. Master.

[67] Taylor, M., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffmann, H., Johnson, P., Lee, J., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., and Agarwal, A. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro 22*, 2 (March/April 2002), 25–35.

[68] Theobald, K., Gao, G., and Hendren, L. On the limits of program parallelism and its smoothability. In *Microarchitecture (MICRO)* (Portland, Oregon, USA, 1992), IEEE Computer Society, pp. 10–19.

[69] Thuresson, M., Själander, M., Björk, M., Svensson, L., Larsson-Edefors, P., and Stenstrom, P. FlexCore: Utilizing exposed datapath control for efficient computing. *Journal of Signal Processing Systems 57*, 1 (April 2008), 5–19.

[70] Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., and Taylor, M. Conservation cores: reducing the energy of mature computations. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Pittsburgh, Pennsylvania, USA, 2010), J. Hoe and V. Adve, Eds., ACM, pp. 205–218.

[71] Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Babb, J., Amarasinghe, S., and Agarwal, A. Baring it all to software: RAW machines. *IEEE Computer 30*, 9 (September 1997), 86–93.

[72] Wall, D. Limits of instruction-level parallelism. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Santa Clara, California, USA, 1991), ACM, pp. 176–188.

[73] Wenisch, T., Wunderlich, R., Ferdman, M., Ailamaki, A., Falsafi, B., and Hoe, J. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro 26*, 4 (2006), 18–31.

[74] Yazdanpanah, F., Alvarez-Martinez, C., Jimenez-Gonzalez, D., and Etsion, Y. Hybrid dataflow/von-Neumann architectures. *IEEE Transactions on Parallel and Distributed Systems 25*, 6 (June 2014), 1489–1509.

[75] Zyuban, V., Brooks, D., Srinivasan, V., Gschwind, M., Bose, P., and Strenski, P. Integrated analysis of power and performance for pipelined microprocessors. *IEEE Transactions on Computers 53*, 8 (August 2004), 1004–1016.

[76] Zyuban, V., and Kogge, P. The energy complexity of register files. In *International Symposium on Low Power Electronics and Design (ISLPED)* (Monterey, CA, USA, 1998), IEEE Computer Society, pp. 305–310.

[77] Äijö, T., Jääskeläinen, P., Elomaa, T., Kultala, H., and Takala, J. Integer linear programming-based scheduling for transport triggered architectures. *ACM Transactions on Architecture and Code Optimization (TACO) 12*, 4 (January 2016), 59:1–59:22.