Technical Report 2008-005
Collapsing the Hierarchy of Parallel Computational Models*

Stefan D. Bruda and Yuanqiao Zhang
Department of Computer Science
Bishop's University
2600 College St
Sherbrooke, Quebec J1M 1Z7, Canada
{bruda|yqzhang}@cs.ubishops.ca

September 27, 2008

Abstract

We investigate the computational power of parallel models with directed reconfigurable buses and with shared memory. Based on feasibility considerations present in the literature, we split these models into "heavyweight" (directed reconfigurable buses, the Combining PRAM, and the BSR) and "lightweight," (all the other PRAMs) and then find that the heavyweight class is strictly more powerful than the lightweight class, as expected. On the other hand, we contradict the long held belief that the heavyweight models (namely, the Combining CRCW PRAM, the BSR, and the models with directed reconfigurable buses) form a hierarchy, showing that all of them are identical in computational power with each other. We start the process by showing that the Collision write conflict resolution rule is universal on models with reconfigurable buses (in the sense that complex conflict resolution rules such as Priority and Combining can be simulated with constant overhead by Collision).

Keywords: parallel computation, real-time computation, reconfigurable multiple bus machine, reconfigurable network, parallel random access machine, broadcast with selective reduction, concurrent-read concurrent-write conflict resolution rules, graph accessibility problem

1 Introduction

The concurrent-read concurrent-write parallel random access machine (CRCW PRAM) is the most convenient model of parallel computation and so it is used extensively in analyzing parallel solutions to various problems. Lower bounds on the PRAM are in particular very strong. The Priority CRCW PRAM is sometimes considered [17] to be at the upper level of feasible parallel models. The broadcast with selective reduction (BSR) [2, 4] on the other hand is at present the most powerful model of parallel computation, with the Combining CRCW PRAM falling somewhere in between. By logical extension of the Priority CRCW PRAM being at the upper end of the feasibility chain, the Combining CRCW PRAM and the BSR should not be considered feasible. The BSR in particular is often frowned upon as too powerful to be practical (even despite the existence of a feasible implementation [3]), but is attractive from the point

*This research was supported by the Natural Sciences and Engineering Research Council of Canada. Part of this research was also supported by Bishop's University.

of view of analysis and design of algorithms. Indeed, fast and efficient algorithms for the BSR have been developed for various problems [14, 15].

Models with directed reconfigurable buses (namely the directed reconfigurable multiple bus machine or DRMBM and the directed reconfigurable network or DRN) have been studied in the past [5, 19], being recognized as realistic models for VLSI design and other parallel machines [6, 13]. While the BSR embraces the combining of values written simultaneously into the same memory location, such a combination is justly disregarded (in as much as little to no work exists on the matter) on bus-based models. Indeed, it is highly questionable how could such a combining operation be implemented on distributed resources such as buses.

It is widely believed (though to our knowledge not proven) that Priority CRCW PRAM is strictly less powerful that the Combining CRCW PRAM, which is in turn strictly less powerful than the BSR. The models with directed reconfigurable buses have been shown to be at least as powerful as the Priority CRCW PRAM [22] but are otherwise not placed anywhere in this hierarchy.

In all, mirroring the beliefs and formal results summarized above, one can identify two "categories" of models of parallel computation: we thus call the Priority CRCW PRAM and the models below it in terms of computational power *lightweight* models, with the *heavyweight* models represented by the Combining CRCW PRAM, the BSR, and the models with directed reconfigurable buses.

The purpose of this paper is then to analyze the relation between these two categories in terms of computational power, as well as the relations between various models within the heavyweight class. Although not always explicit, we always have in mind real-time computations, so we are using a strong notion of relationship: we say that model A is more powerful than model B only if $t(n)$ computational steps of model B using polynomial resources can be simulated in $O(t(n))$ steps of model A using polynomial resources. We often accomplish this by showing how one model is capable of simulating one computational step of the second model in constant time (and the other way around whenever we want to prove equality).

We start our analysis however on the reconfiguration side. Previous work [7, 8] completed the characterization of models with directed reconfigurable buses running in constant time. Between other things, it was found that the Collision conflict resolution rule is universal for these models, meaning that it can simulate the other powerful (and likely unfeasible) conflict resolution rules. We show here that the universality of Collision on directed reconfigurable buses is not restricted to the constant time case, but happens for any running time.

We then use the result mentioned in the above paragraph as both motivation and tool to offer an analysis of computational power for various models of parallel computation. As expected, we find that the class of heavyweight models is strictly more powerful than the class of lightweight models. Surprisingly, we also find that all the heavyweight models are however equivalent with each other, despite the perceived high computational power of the BSR.

Our results are rather significant, as we essentially free the analysis of parallel algorithms and problems from a number of restrictions such as whether using the powerful Broadcast instruction of the BSR diminishes the practicality of the analysis (specifically, we are not aware of any real shared memory machine that implements Broadcast, yet as a consequence of our result such an instruction can be used with no consequences in the design of algorithms for such machines), or whether using the Combining resolution rule on distributed resources like buses diminishes the practicality of the analysis. At the same time, we offer a strict delimitation between the two classes of heavyweight and lightweight models of parallel computation.

These results are in particular significant from the point of view of the analysis and design of algorithms on models with reconfigurable buses: For one thing, we make the whole discussion of whether Combining is feasible irrelevant, and we allow the analysis and design to use Combining as conflict resolution rule with the knowledge that the implementation can then be realized using the easily implementable Collision rule

instead. Secondly, we show that the analysis and design of algorithms on models with reconfigurable buses does not need to be done directly on that model; indeed, we show that one can use the possibly more convenient (and certainly powerful) BSR for this purpose, with the knowledge that any BSR algorithm or analysis is directly applicable to the target model (with reconfigurable buses).

Furthermore, these results become particularly significant in conjunction with previous work on real-time parallel computations [8]. We show that the Combining CRCW PRAM and the BSR are as powerful as reconfigurable buses (thus being useful for the design and analysis of VLSI circuits). We also show that the Combining CRCW PRAM is the simplest model that can solve exactly all the problems solvable under no matter how tight real-time constraints. In effect, we "simplify" the domain of real time, thus strengthening previous results on real-time computations.

The paper continues as follows: We present in the next section the necessary preliminaries, including the previously obtained characterization of constant time computations on directed reconfigurable buses. The universality of the Collision resolution rule is established in Section 3. The characterization of heavyweight and lightweight models is the subject of Section 4, which is followed by some generalizations, including real-time considerations in Section 5. We conclude in Section 6.

2 Preliminaries

Results proved elsewhere are introduced henceforth as Propositions, whereas results proved in this work are introduced as Theorems. Intermediate results are all Lemmata.

Given a directed graph $G = (\{1, 2, ..., n\}, E)$ (expressed for instance by an adjacency matrix), GAP$_{i,j}$ is the problem of determining whether vertex j is accessible from vertex i. Given n integer data x_1, x_2, ..., x_n, the problem of computing the function PARITY$_n(x_1, x_2, \ldots, x_n) = (\sum_{i=1}^{n} x_i) \bmod 2$ is denoted by PARITY$_n$. L [NL] is the set of languages that are accepted by deterministic [nondeterministic] Turing machines that use at most $O(\log n)$ work space on any input of size n [18]. By standard abuse of notation we also use L and NL for problems (rather than languages).

For some language $L \in$ NL there exists a nondeterministic Turing machine $M = (K, \Sigma, \delta, s_0)$ that accepts L and uses $O(\log n)$ work space. K is the set of states (not containing the halt state h), Σ is the tape alphabet (we can assume without loss of generality that $\Sigma = \{0, 1\}$), δ is the transition relation, and s_0 is the initial state. M accepts an input string x if and only if M halts on x. A configuration of M working on input x is defined as a tuple (s, i, w, j), where s is the state, i and j are the positions of the heads on the input and work tapes, respectively, and w is the content of the work tape. There are therefore $poly(n)$ possible configurations of M. For two configurations v_1 and v_2, we write $v_1 \vdash v_2$ if and only if v_2 can be obtained by applying δ exactly once on v_1 [18]. The set of possible configurations of M working on x forms a directed graph $G(M, x) = (V, E)$ as follows: V contains one vertex for each and every possible configuration of M working on x, and $(v_1, v_2) \in E$ if and only if $v_1 \vdash v_2$; then $x \in L$ if and only if some configuration (h, i_h, w_h, j_h) is accessible in $G(M, x)$ from the initial configuration. For any language $L \in$ NL and any x, determining whether $x \in L$ can thus be reduced to the problem GAP$_{1,|V|}$ for $G(M, x) = (V, E)$, where M is an NL Turing machine that decides L.

The class of problems in NL with the addition of real-time constraints is denoted by NL/rt [8]. rt-PROC$^M(f)$ denotes the class of those problems solvable in real time by the parallel model of computation M that uses $f(n)$ processors (and $f(n)$ buses if applicable) for any input of size n. The following strongly supported conjecture is then established.

Claim 1 [8] rt-PROC$^{\text{CRCW F-DRMBM}}(poly(n)) =$ NL/rt.

Two main models with reconfigurable buses have been developed in the literature: the *reconfigurable network* (RN for short) [5, 20] and the *reconfigurable multiple bus machine* (or RMBM) [19, 20]. Shared memory models are more often than not represented by the *parallel random access machine* (or PRAM) [2, 17] and by its extension, the *broadcast with selective reduction* (or BSR) [4].

2.1 The PRAM and the BSR

A PRAM [2, 17] consists in a number of processors that share a common random-access memory. The processors execute the instructions of a parallel algorithm synchronously. The shared memory stores intermediate data and results, and also serves as communication medium for the processors. The model is further specified by defining the memory access mode; we thus obtain exclusive-read exclusive-write (EREW), concurrent-read exclusive-write (CREW) and concurrent-read concurrent-write (CRCW) PRAM. While reading concurrently from the shared memory is defined straightforwardly, writing concurrently into the shared memory requires the introduction of a conflict resolution rule (for the case in which two or more processors write into the same memory location). Four such conflict resolution rules are in use: Common (the processors writing simultaneously in the same memory location must write the same value or else the algorithm fails), Collision (multiple processors writing into the same memory location garble the result so that a special "collision" marker ends up written at that location instead of any processor-provided data), Priority (processors are statically numbered and the memory location receives the value written by the lowest numbered processor), and Combining (where a binary, associative reduction operation is performed on all the values sent by all the processors to the same memory location and the result is stored in that memory location).

Given the obviously increased computational power as well as the straightforward implementation of concurrent-read machines, we will not consider exclusive-read variants. For similar reasons, exclusive-write machines will receive a spotty consideration (if any).

The BSR model [2, 3, 4] is an extension of the Combining CRCW PRAM. All the read and write operations of the Combining CRCW PRAM can also be performed by the BSR. In addition, all the BSR processor can write simultaneously into all the memory locations (the Broadcast instruction). Every Broadcast instruction consists in three steps: In the *broadcasting step*, all the n participating processors produce a datum d_i and a tag g_i, $1 \leq i \leq n$, destined to all the m memory locations. In the *selection step* each of the m memory locations uses a limit l_j, $1 \leq j \leq m$ and a selection rule $\sim \in \{<, \leq, =, \geq, >, \neq\}$ to test the received data; the datum d_i is selected for the next step if and only if $g_i \sim l_j$. Finally, the *reduction step* combines all the data d_i destined for memory location j, $1 \leq j \leq m$ and selected in the previous step using a binary, associative operator $\mathcal{R}$, and then writes the result into memory location j. The Broadcast instruction is performed simultaneously for all the processors and all the memory locations.

Typically, the reduction operator $\mathcal{R}$ of the BSR as well as the Combining operator of the Combining CRCW PRAM can be any of the following operations: Σ (sum), Π (product), $\bigwedge$ (logical conjunction) $\bigvee$ (logical disjunction), $\bigoplus$ (logical exclusive disjunction), $\max$ (maximum), and $\min$ (minimum).

We denote by X CRCW PRAM$(p(n), t(n))$ the class of problems of size n that are solvable in time $t(n)$ on a CRCW PRAM that uses $p(n)$ processors and X as collision resolution rule, $X \in \{\text{Common}, \text{Collision}, \text{Priority}, \text{Combining}\}$. Similarly, we denote by BSR$(p(n), t(n))$ the class of problems of size n that are solvable in time $t(n)$ on a BSR that uses $p(n)$ processors.

The notion of uniform families of PRAM or BSR machines exists [16] and is similar to the notion of uniformity for models with reconfigurable buses (to be introduced below). However, we do not need such a notion in this paper. Still, we assume as customary that one location in the shared memory has $\log n$ size for an input of size n, and that the number of memory locations are upper bounded by a polynomial in the

number of processors used. Again as usual, we assume that every BSR or PRAM processor has a constant number of internal registers, each of size $\log n$. Finally, we assume that every PRAM processor knows its number as well as the total number of processors in the system.

We will use the following results regarding the PRAM and the BSR.

Proposition 2.1 [9] PARITY$_n \notin$ Priority CRCW PRAM$(poly(n), O(1))$.

Proposition 2.2 [14] *The reflexive and transitive closure of a graph with n vertices (and thus the graph accessibility problem) is in* BSR$(n^2, O(1))$.

2.2 The RMBM and the RN

An RMBM [19] consists of a set of p *processors* and b *buses*. For each processor i and bus b there exists a *switch* controlled by processor i. Using these switches, a processor has access to the buses by being able to read or write from/to any bus. A processor may be able to *segment* a bus, obtaining thus two independent, shorter buses, and it is allowed to *fuse* any number of buses together by using a *fuse line* perpendicular to and intersecting all the buses. DRMBM, the *directed* variant of RMBM, is identical to the undirected model, except for the definition of fuse lines: Each processor features two fuse lines (*down* and *up*). Each of these fuse lines can be electrically connected to any bus. Assume that, at some given moment, buses i_1, i_2, ..., i_k are all connected to the down [up] fuse line of some processor. Then, a signal placed on bus i_j is transmitted in one time unit to all the buses i_l such that $i_l \geq i_j$ [$i_l \leq i_j$]. If some RMBM [DRMBM] is not allowed to segment buses, then this restricted variant is denoted by F-RMBM [F-DRMBM] (for "fusing" RMBM/DRMBM). The *bus width* of some RMBM (DRMBM, etc.) denotes the maximum size of a word that may be placed (and read) on (from) any bus in one computational step.

For CRCW RMBM, the most realistic conflict resolution rule is Collision , where two values simultaneously written on a bus result in the placement of a special, "collision" value on that bus. We consider for completeness other conflict resolution rules such as Common, Priority, and even Combining. However, we find that all of these rules are in fact equivalent to the seemingly less powerful Collision rule (Theorem 3.1). We restrict only the Combining mode, requiring that the combining operation be associative and computable in nondeterministic linear space (this property clearly holds for any reasonable such an operation).

An RMBM (DRMBM, F-DRMBM, etc.) *family* $R = (R_n)_{n\geq 1}$ is a set containing one RMBM (DRMBM, etc.) construction for each $n > 0$. A family R solves a problem π if R_n solves all inputs of π of size n. We say that some RMBM family R is *uniform* if there exists an NL Turing machine M that, given n, produces the description of R_n using $O(\log(p(n)b(n)))$ work tape cells. We henceforth drop the "uniform" qualifier, with the understanding that any RMBM family described here is uniform. If some family $R = (R_n)$ solves a problem π, and each R_n, $n > 0$, uses $p(n)$ processors, $b(n)$ buses, X as write conflict resolution rule ($X \in \{\text{CREW}, \text{Common CRCW}, \text{Collision CRCW}, \text{Priority CRCW}, \text{Combining CRCW}\}$), and runs in $t(n)$ time, then we say that $\pi \in X$ RMBM$(p(n), b(n), t(n))$ (or $\pi \in X$ F-DRMBM$(p(n), b(n), t(n))$, etc.), and that R has *size complexity* $p(n)b(n)$ and *time complexity* $t(n)$. Whenever we state a property that holds for any conflict resolution rule we drop X (thus writing $\pi \in$ RMBM$(p(n), b(n), t(n))$, etc.).

It should be noted that a directed RMBM can simulate an undirected RMBM by simply keeping all the up and down fuse lines synchronized with each other.

An RN [5] is a network of processors representable as a connected graph whose vertices are the processors and whose edges represent fixed connections between processors. Each edge incident to a processor corresponds to a (bidirectional) port of the processor. A processor can partition its ports such that all the

ports in the same block of the partition are electrically connected (fused) together. The edges that are thus connected together form a bus. CREW, Common CRCW, Collision CRCW, etc. are defined as for the the RMBM model. The *directed* RN (DRN) is similar to the RN, except that the edges are directed. The concept of (uniform) RN family is identical to the RMBM concept. For some write conflict resolution rule X ($X \in \{\text{CREW}, \text{Common CRCW}, \text{Collision CRCW}, \text{Priority CRCW}, \text{Combining CRCW}\}$), X RN$(p(n), t(n))$ [X DRN$(p(n), t(n))$] is the set of problems solvable by RN [DRN] uniform families with $p(n)$ processors ($p(n)$ is also called the *size complexity*) and $t(n)$ running time using the conflict resolution rule X. We drop X when the stated property refers to any conflict resolution rule, as we do in the RMBM case.

As in the RMBM case, we note that for any undirected RN there exists a directed RN with the same size complexity and the same running time that simulates it (the directed RN provides a couple of edges of opposite directionality for every edge in the undirected RN).

2.3 Reconfiguration and Small Space

The characterization of constant time DRMBM and DRN computations mentioned in the introduction of this paper can be summarized as follows:

Proposition 2.3 [7, 8] *For any* $n \in \mathbb{N}$,

1. DRN$(poly(n), O(1))$ = DRMBM$(poly(n), poly(n), O(1))$ = NL = Collision CRCW F-DRMBM$(poly(n), poly(n), O(1))$ *with bus width* 1 = Collision CRCW DRN$(poly(n), O(1))$ *with bus width* 1.

2. *For any* $X \in \{\text{CRCW}, \text{CREW}\}$, $Y \in \{\text{D}, \varepsilon\}$, $Z \in \{\text{RN}(poly(n), O(1)), \text{RMBM}(poly(n), poly(n), O(1))\}$ *and for any write conflict resolution rule, it holds that* $X\,Y\,Z \subseteq$ Collision CRCW Z *with bus width 1.*

3. *For any problem* π *solvable in constant time by some (directed or undirected)* RMBM *family using* $poly(n)$ *processors and* $poly(n)$ *buses,* $\pi \in$ Collision CRCW F-DRMBM$(poly(n), poly(n), O(1))$ *with bus width* 1.

While further studying the Collision rule we shall make use of two intermediate results used to prove Proposition 2.3. To make this paper self contained we include below a sketch of the proof of the first result (but we only state the second result, which has a similar proof).

Lemma 2.4 [8] $\forall n \in \mathbb{N}$*:* CRCW DRMBM$(poly(n), poly(n), O(1)) \subseteq$ NL, *for any write conflict resolution rule and any bus width.*

Proof. Consider some directed RMBM $R \in$ CRCW DRMBM$(poly(n), poly(n), O(1))$ performing step d of its computation ($d \leq O(1)$). We find an NL Turing machine M_d that generates the description of R after step d using $O(\log n)$ space, and thus [10] an NL Turing machine M'_d that receives n' (the number of processors in R) and some $1 \leq i \leq n'$, and outputs the ($O(\log n)$ long) description for processor i instead of the whole description. We establish the existence of M_d and M'_d by induction over d.

M_0 exists by the definition of a uniform family. For each processor p_i and each bus k read by p_i during step d, M_d behaves as follows: M_d maintains two words b and ρ, initially empty. M_d determines for every p_j whether p_j writes on bus k. M_d needs to perform GAP$_{j,i}$ (NL-complete [18] and thus computable in

nondeterministic $O(\log n)$ space) for this step. The local bus configurations at each processor (that is, the edges of the graph for GAP$_{j,i}$) are obtained by calls to M'_{d-1}.

If p_j writes on bus k, then M_d uses M'_{d-1} to determine the value v written by p_j , and updates b and ρ as follows: (a) If b is empty, then it is set to v, and ρ is set to j. (b) If R uses the Collision rule, the collision signal is placed in b. (c) If the conflict resolution rule is Priority, ρ and j are compared; if the latter denotes a higher priority, then b is set to v and ρ is set to j, otherwise, neither b nor ρ are modified; the Common rule is handled similarly. (d) If R uses the Combining resolution rule with operation $\circ$, b is set to the result of $b \circ v$ (noting that the operation $\circ$ is associative and so the resulting value eventually written on the bus is correct; the operation can be performed in $O(\log n)$ space, since the length of both b and v is $O(\log n)$, and $\circ$ is computable in linear space).

Once the content of bus k is determined, the configuration of p_i is updated accordingly, b and ρ are reset, and the same computation is performed for the next bus and/or for the next processor. ■

Lemma 2.5 [7] *For any $X \in \{\text{CRCW}, \text{CREW}\}$, $Y \in \{\text{D}, \varepsilon\}$, $n \in \mathbb{N}$, and for any write conflict resolution rule, it holds that $X\ Y\text{RN}(poly(n), O(1)) \subseteq$ Collision CRCW DRN$(poly(n), O(1))$.*

3 Collision Is Universal On Directed Reconfigurable Buses

The results regarding constant time computations on directed reconfigurable buses can be used to extend the universality of the Collision conflict resolution rule to any running time.

Theorem 3.1 *The* Collision *conflict resolution rule is universal on models with directed reconfigurable buses; that is: For any $X \in \{\text{CRCW}, \text{CREW}\}$, $Y \in \{\text{D}, \varepsilon\}$, $Z \in \{\text{RN}(poly(n), \cdot), \text{RMBM}(poly(n), poly(n), \cdot)\}$, and $t : \mathbb{N} \to \mathbb{N}$ it holds that $\forall n \in \mathbb{N} : X\ Z(t(n)) \subseteq$ Collision CRCW D$Z(O(t(n)))$.*

Proof. The proof is immediate for CREW machines, so we only consider CRCW machines. We will first replace such a machine with a CREW machine and then we use GAP computations to accomplish the result of the original concurrent write operations.

Let R be an RMBM family in CRCW DRMBM$(poly(n), poly(n), t(n))$, and recall the NL machines M'_d constructed in the proof of Lemma 2.4. We then take the original R, replace its conflict resolution rule with Collision, and then split every step i of the computation of R into a constant number of steps, as follows:

1. Each processor p of R reads the content of the original buses as required and then performs the prescribed computation for step i, except that whenever p wants to write to bus k it also writes the same value of a dedicated bus k_p (there is one such a bus for each processor).

2. A suitably modified machine M'_d from the proof of Lemma 2.4 (call this machine M) computes the content of all the original buses of the network, based on the configurations of all the processors and the content of the (new) k_p buses; the content of bus k thus computed is placed on a brand new bus k'. The meaning of "suitably modified" is that the machine does not need to determine the configurations of the processors of R (that is, how they fuse and segment buses); indeed, these configurations are already present in the processors themselves, so the machine starts directly with GAP computations to determine the content of the buses, as described in the proof of Lemma 2.4.

3. A designated processor p_k transfers the content of bus k' onto bus k and the algorithm continues with step $i + 1$.

Given that M is an NL Turing machine, it can be implemented by a polynomially bounded DRMBM R_M that runs in constant time, so the modified step i takes $O(1)$ time (and then the whole computation takes $O(t(n))$ time, as desired). This new RMBM needs to read the configurations of R; for this purpose a polynomial number of new buses can interconnect each processor of R with each processor of R_M. We end up with a polynomial number of buses; that we have a polynomial number of processors is immediate. The correctness of the transformation follows from Lemma 2.4 and Proposition 2.3. The proof is complete.

The proof for RN represents a minor variation of the above proof in light of Lemma 2.5 (that replaces Lemma 2.4) and of Proposition 2.3. ■

4 The Heavyweight and Lightweight Classes of Parallel Models

Recall that we called Priority CRCW PRAM and all the models of less computational power lightweight, while the Combining CRCW PRAM, the BSR, and the models with directed reconfigurable buses were called heavyweight. We show in this section that all the heavyweight models have the same computational power, and that they are strictly more powerful than the lightweight models:

Theorem 4.1 *For any $n \in \mathbb{N}$ and for any $X \in \{\text{Collision}, \text{Common}\}$,*

$$X \text{ CRCW PRAM}(poly(n), O(t(n))) \subseteq \text{Priority CRCW PRAM}(poly(n), O(t(n))) \subsetneq$$
$$\text{Combining CRCW PRAM}(poly(n), O(t(n))) = \text{BSR}(poly(n), O(t(n))) =$$
$$\text{DRMBM}(poly(n), poly(n), O(t(n))) = \text{DRN}(poly(n), O(t(n))).$$

Proof. Theorem 4.1 is a direct consequence of Lemmata 4.2 to 4.7 below. Specifically, all the inclusions shown in the theorem are proven in the mentioned Lemmata one by one.

That $\text{DRMBM}(poly(n), poly(n), O(t(n))) = \text{DRN}(poly(n), O(t(n)))$ is shown elsewhere [19]. ■

We complete all the proofs below by showing how the model on the right hand side of the inclusion simulates in constant time one computational step of the model on the left hand side. Once this is shown, the inclusion that needs to be proven becomes immediate.

We note that some of the hierarchy analyzed here has also been investigated earlier [12]. In particular, Lemma 4.2 has been established in a stronger version [12] (namely, that the two models are equivalent). For completeness however we include here our proof of this result.

Lemma 4.2 $\forall n \in \mathbb{N}$: Collision CRCW PRAM$(poly(n), O(t(n))) \subseteq$ Priority CRCW PRAM$(poly(n), O(t(n)))$.

Proof. A Collision CRCW PRAM with k processors p_i, $1 \leq i \leq k$ and m memory locations u_j, $1 \leq j \leq m$ is readily simulated by a Priority CRCW PRAM with $2k + m$ processors denoted by $p_i^{\downarrow}$ ($1 \leq i \leq k$), $p_i^{\uparrow}$ ($1 \leq i \leq k$), and p_j^m ($1 \leq j \leq m$). (Note however that the processor group p_j^m and the processor group $p_i^{\downarrow}$ plus $p_i^{\uparrow}$ take turns in the simulation, so the actual number of processors required is $\max(2k, m)$; however, the explicit differentiation eases the presentation.)

In addition to the original memory locations u_j, we use two more "banks" of the same size $u_j^{\downarrow}$ and $u_j^{\uparrow}$, $1 \leq j \leq m$. A Collision CRCW PRAM step (read, compute, write) is then simulated as follows:

1. For every $1 \leq i \leq k$, both the processors $p_i^{\downarrow}$ and $p_{k+1-i}^{\uparrow}$ perform the same read, compute, and write cycle as the original p_i, with the following addition: Whenever processor $p_i^{\downarrow}$ writes into memory

location u_j, it also writes its number j into memory location $u_j^{\downarrow}$; similarly, whenever processor $p_{k+1-i}^{\uparrow}$ writes into memory location u_j, it also writes its number $k+1-i$ into memory location $u_j^{\uparrow}$.

2. Every processor p_i^m writes the collision value into memory location u_j if and only if $u_j^{\downarrow} \neq u_j^{\uparrow}$.

That the above simulation takes constant time is immediate. Note further that after Step 1 of the simulation the location $u_j^{\downarrow}$ [$u_j^{\uparrow}$] contains the index of the lowest [highest] ranked original processor (p_i)that modified the memory location u_j (indeed, we operate on a Priority CRCW model, so only the lowest numbered processor succeeds in writing into a given memory location; we then chose the processors numbers in appropriate manner for the desired property to happen). Then, whenever $u_j^{\downarrow} \neq u_j^{\uparrow}$, more than one processor wrote into the given memory location, so a collision occurred. Step 2 places collision markers accordingly. The simulation is complete. ■

Lemma 4.3 $\forall n \in \mathbb{N}$: Common CRCW PRAM($poly(n), O(t(n))) \subseteq$ Priority CRCW PRAM($poly(n)$, $O(t(n))$).

Proof. We simulate now a computational step of a Common CRCW PRAM with k processors and m memory locations in constant time using a Priority CRCW PRAM. The simulation will use the same number k of processors (we denote them by p_i, $1 \leq i \leq k$) and the same memory space (denoted by u_j, $\leq j \leq m$). The simulation proceeds as follows:

1. All the processors p_i carry on the computational step prescribed by the Common CRCW PRAM algorithm, including the operation of writing into the shared memory (recall however that we are now using the Priority conflict resolution rule).

2. Each processor p_i that wrote a value v_i into memory location u_j in Step 1 also remembers v_i by storing it into an otherwise unused internal register ρ.

3. Every p_i that wrote a value into location u_j in Step 1 read the content of u_j and compares it with the content of its register ρ.

   (a) If the contents of u_j and ρ are the same then either (a) p_i is the sole processor which wrote into u_j, (b) p_i is the highest priority processor which wrote into u_j, or (c) p_i and the highest priority processor agree on the value written into u_j. None of these cases violate the Common resolution rule, so p_i does not do anything.

   (b) If on the other hand u_j and ρ contain different values, then the value written into u_j by p_i disagrees with the value written in the same location by some other processor, which in turn violates the Common resolution rule. So p_i aborts the algorithm and reports failure.

Note that in effect we chose *one* of the processors writing concurrently into a memory location as representative for all the others (given that we have a Priority machine at our disposal, that representative turned out to be the processor with the highest priority; however the way we chose a representative is immaterial). Every processor that wants to write a value in some memory location compares now its value with the value already written by its representative; if the value is different, then the Common conflict resolution rule is violated; otherwise all is good and the overall algorithm continues with the next step.

The above proof uses the usual definition of Common, as presented in Section 2. Still, we note that sometimes this definition is termed "Fail-safe Common," case in which the "Fail Common" or "Tolerant"

variant is also defined [2, 11]. In such a variant, any computational step that violates the Common resolution rule is discarded completely (that is, for all the processors in the system) and the algorithm continues with the next step (instead of aborting the computation). A proof for this modified variant of Common is readily possible. Indeed, all we need is "backup" copies of the memory locations used by the algorithm, plus one memory location used to signal any violation to everybody. The processors now use the backup memory to perform all the simulation described above, except that they set the violation flag instead of aborting the algorithm whenever a violation of the Common resolution rule occurs (since is is just a flag, using Priority as conflict resolution rule will do just as well as almost any other rule). At the end of the computational step, all the processors inspect the flag and write again the values they wanted to write in the first place (this time in the main memory, not the backup) only if the flag is not set; otherwise they all proceed to the next step immediately, thus leaving the main working memory unchanged. ■

Lemma 4.4 $\forall n \in \mathbb{N}$: Priority CRCW PRAM$(poly(n), O(t(n))) \subsetneq$ Combining CRCW PRAM$(poly(n), O(t(n)))$.

Proof. A Priority CRCW PRAM with k processors p_i, $1 \leq i \leq k$ and m memory locations u_j, $1 \leq j \leq m$ is readily simulated by a Combining CRCW PRAM with the same number of processors (denoted by p'_i) and $2m$ memory locations (denoted by u_i and u'_i):

1. Each processor p'_i performs the same read and compute operations as p_i. Instead of writing (to memory location u_j), p'_i however performs a "dry run" by writing its number into memory location u'_j using a Combining CRCW write with $\min$ as combining operation.

2. Each processor p'_i performs now the real write operation: It writes into the memory location u_j in which it wanted to write to begin with, but does so only if its number matches the value stored in u'_j. The $\min$ as combining operation performed over the locations u'_j in the previous step ensures that a matching occurs only for the lowest numbered processor, as desired.

That Combining CRCW PRAM$(poly(n), O(t(n))) \neq$ Priority CRCW PRAM$(poly(n), O(t(n)))$ is an immediate extension of Proposition 2.1. Indeed, by Proposition 2.1 PARITY$_n$ is not solvable in constant time on a Priority CRCW PRAM with polynomial number of processors. On the other hand, PARITY$_n$ is trivially solvable in constant time using a Combining CRCW PRAM with n processors: Every processor p_i of such a machine, $1 \leq i \leq n$ holds one input datum x_i and writes it into some designated memory location μ by performing a Combining CRCW using Σ as combining operation; then p_1 performs a modulo operation on μ and thus μ contains the output of PARITY$_n$ for the given instance, as desired. ■

Lemma 4.5 $\forall n \in \mathbb{N}$: Combining CRCW PRAM$(poly(n), O(t(n))) =$ BSR$(poly(n), O(t(n)))$.

Proof. That Combining CRCW PRAM$(poly(n), t(n)) \subseteq$ BSR$(poly(n), O(t(n)))$ is immediate from the definition of the BSR. Surprisingly enough, the reverse inclusion is also true. We show now this reverse inclusion. Specifically, we show how one BSR computational step is simulated by a Combining CRCW PRAM in constant time.

Consider a BSR with k processors and m memory locations. Every BSR processor p_i is simulated by a set of m PRAM processors r_{ij}, $1 \leq j \leq m$. The PRAM memory is doubled, every BSR memory location u_j will be simulated by two PRAM memory locations u_j^d and u_j^l, $1 \leq j \leq m$. Finally, the PRAM uses extra processors p_j^u, $1 \leq j \leq m$ (once more processors p_j^u and r_{ij} actually take turns in the simulation so we differentiate them for convenience only; the actual number of processors used is in fact smaller as these two groups can overlap with each other). The PRAM simulation of a BSR Broadcast step (read, compute, Broadcast) then proceeds as follows:

- *Read and Compute:* For $1 \leq i \leq n$, all the processors r_{ij}, $1 \leq j \leq m$ perform the reading and the computation prescribed for p_i. Every time some processor wants to read the value of u_j it will read the value of u_j^d instead. Processors r_{ij} will then *all* hold the values of the datum d_i and the tag g_i originally computed by p_i.

  Note that there are n groups of processors; all the m processors in the same group perform the same computation. This may appear wasteful but is intentional and will be used to simulate the Broadcast instruction.

- *Selection limits:* Every processor p_j^u computes the limit l_j associated with u_j in the selection phase of the BSR step, and stores it in the memory location u_j^l.

- *Broadcast instruction:* r_{ij} will be responsible for the data written by the BSR processor p_i into memory location r_j:

  1. r_{ij} reads l_j from memory location u_j^l so that it holds d_i, g_i, and l_j;
  2. r_{ij} then computes the selection criterion $g_i \sim l_j$ as prescribed by the BSR algorithm;
  3. r_{ij} writes d_j into memory location u_j^d if and only if $g_i \sim l_j =$ True, using a Combining CRCW write with the combining operator prescribed by the BSR algorithm.

  Note that for some fixed i all the processors r_{ij}, $1 \leq j \leq m$ contain identical data, so p_i's replacement covers all the memory locations, thus realizing the desired broadcast.

In effect, we use one PRAM processor for every pair processor–memory location in the BSR algorithm. This allows for an immediate simulation of the broadcast phase of a Broadcast instruction: Instead of broadcasting, every PRAM processor is now responsible for writing to one memory location only; but then since we have as many processors as memory locations, we nonetheless write to all the memory locations at once, as desired. The correctness of the rest of the simulation is immediate, and so is the overall constant running time. ■

Lemma 4.6 $\forall n \in \mathbb{N}$: BSR$(poly(n), t(n)) \subseteq$ DRMBM$(poly(n), poly(n), O(t(n)))$.

Proof. We are given a BSR with k processors and m memory locations. Without loss of generality we build a Combining CRCW DRMBM that simulates the given BSR; for indeed, once such a construction is established a Collision CRCW DRMBM with polynomially bounded resources and $O(t(n))$ running time is an immediate consequence of Theorem 3.1.

Our DRMBM is depicted in Figure 1. It has $(k+1) \times m$ processors, which we denote by $p_{i,j}$, $1 \leq i \leq k+1$, $1 \leq j \leq m$. The "real" processors $p_{i,1}$, $1 \leq i \leq k$ perform identically to the processors of the original BSR (except for the bus manipulation routines). The "memory" processors $p_{k+1,j}$, $1 \leq j \leq m$ simulate the locations of the shared memory. They designate one register μ that will hold the data stored in the respective memory location.

As shown in the figure, the DRMBM features $k+m$ buses denoted by D_i, $1 \leq i \leq k$ and L_j, $1 \leq j \leq m$. In addition, every "memory" processor $p_{k+1,j}$ has a dedicated bus M_j, $1 \leq j \leq m$ (not shown in the figure). The DRMBM simulates one step of the BSR computation (that is, one read-compute-broadcast-select-reduce cycle) in constant time as follows (selected steps are shown as circled numbers in the figure):

1. Every "memory" processor $p_{k+1,j}$ puts the datum held in its designated register μ on bus M_j. Every "real" processor $p_{i,1}$ that is interested in some memory data reads the bus of interest.

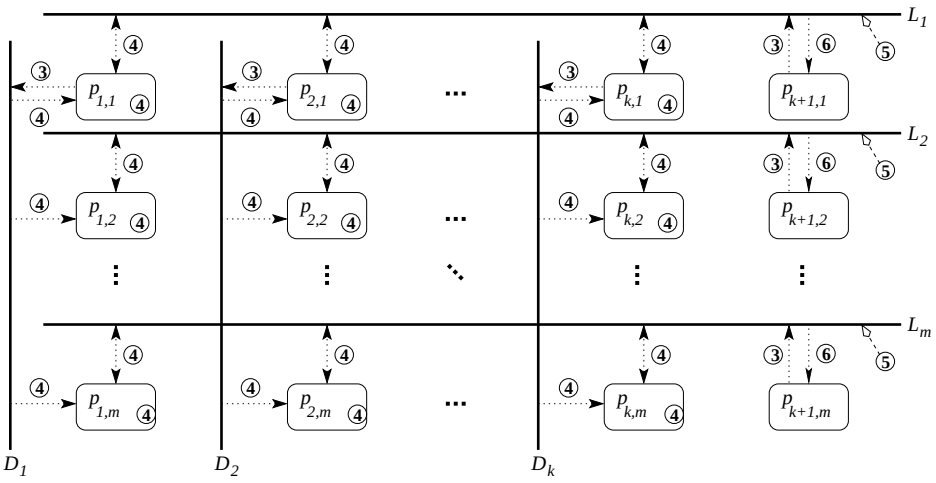

Figure 1: A DRMBM simulation of the BSR

2. All the "real" processors $p_{i,1}$ perform the computation prescribed by the BSR algorithm.

3. Every "real" processor $p_{i,1}$ broadcasts the computed pair (d_i, g_i) by putting it on bus D_i. Every "memory" processor $p_{k+1,j}$ computes and broadcasts its limit l_j by placing it on bus L_j.

4. The processors $p_{i,j}$, $1 \leq i \leq k$, $1 \leq j \leq m$ implement now the selection phase: $p_{i,j}$ reads the pair (d_i, g_i) and the limit l_j from the buses D_i and L_j, respectively. It then computes $g_i \sim l_j$ and places d_i on bus L_i as appropriate (if and only if $g_i \sim l_j =$ True).

5. The reduction is accomplished by the buses L_j, $1 \leq j \leq m$ which perform the combining (reduction) operation prescribed by the BSR algorithm. (Recall that we use Combining without loss of generality, as we can then convert the Combining RMBM into a Collision RMBM using Theorem 3.1.)

6. Finally, every "memory" processor $p_{k+1,j}$ reads bus L_j and stores the datum thus obtained into its designated register μ.

It is immediate that the above steps complete in constant time and accomplish the desired computation. This sequence of steps is applied repeatedly for every step performed by the BSR. The running time of the whole simulation is then $O(t(n))$, as desired. We have $(k+1) \times m = poly(n)$ processors and $2m + k = poly(n)$ buses, so the DRMBM uses polynomial resources. The proof is complete. ■

Lemma 4.7 $\forall n \in \mathbb{N}$: DRMBM$(poly(n), poly(n), t(n)) \subseteq$ BSR$(poly(n), O(t(n)))$.

Proof. The proof of this result relies as expected on Proposition 2.2: The capability of the BSR to compute the reflexive and transitive closure (or equivalently GAP) in constant time allows this model to simulate bus fusing, which is the only essential supplementary capability of the RMBM over the BSR. Indeed, we use the same technique we used in the proof of Theorem 3.1, in that we simulate first a DRMBM that does not fuse buses, and then we combine the content of the (unfused) buses using GAP computations.

As before, we complete the proof by showing how a DRMBM computational step can be simulated in constant time by a BSR. By Proposition 2.3, a proof for the Collision F-DRMBM suffices.

Consider then a Collision F-DRMBM with k processors and m buses. The BSR that simulates it uses k processors p_i, $1 \leq i \leq k$ to simulate the DRMBM processors and m^2 processors (referred to collectively as P^c and individually as p^c_{ij}, $1 \leq i, j \leq m$) dedicated to the computation of the reflexive and transitive closure of an $m \times m$ graph. In terms of memory space, the notable areas include m memory locations b_j and another m memory locations c_j, $1 \leq j \leq m$ to simulate the buses, $m \times m$ memory locations G_{ij}, $1 \leq i, j \leq m$ to hold the connectivity graph for the buses, and another $m \times m$ memory locations $C_{i,j}$, $1 \leq i, j \leq m$ to hold the closure of the aforementioned graph. We assume that there exists a value never placed on a bus by any DRMBM processor (the collision marker).

A computational step of a DRMBM processor consists in the following phases: read from the buses, perform the prescribed computation (thus determining what buses to fuse and what to write to buses), fuse buses, write to buses. The BSR simulation of one DRMBM step proceeds then as follows:

1. The P^c processors initialize in parallel G_{ij} to False and c_j to 0, $1 \leq i, j \leq m$
2. Every p_i reads the memory locations b_j as prescribed by the DRMBM algorithm (we replace reading buses with reading memory locations).
3. Every p_i performs the operations prescribed by the DRMBM algorithm.
4. Every p_i that wants to fuse buses in the DRMBM algorithm broadcasts True to all the memory location G_{uv} corresponding to buses u and v being fused. The writing is a Combining CRCW write with $\bigvee$ as combining operator.
5. The locations G_{uv}, $1 \leq u, v \leq m$ now hold the adjacency matrix of the graph of connected buses. The processors P^c then compute the closure of G, putting the result in $C_{i,j}$, $1 \leq i, j \leq m$.
6. Every processor p_i that wants to write some value to bus j writes the corresponding value into b_j using a Combining CRCW write with any combining operator, and writes 1 into memory location c_j using a Combining CRCW write with Σ as combining operator.
7. The processors P^c now alter the content of b_j and c_j as follows: If $C_{ji} =$ False, then p^c_{ji} does not do anything. Otherwise, p^c_{ji} reads the value from b_j and writes it into the memory location b_i using a Combining CRCW write with any combining operator. p^c_{ji} also reads the value from c_j and writes it into the memory location c_i using a Combining CRCW write with Σ as combining operator.
8. Every p^c_{j1}, $1 \leq j \leq m$ reads c_j and places the collision marker into memory location b_j if and only if $c_j > 1$.

Reading data from the buses and performing the prescribed computation are simulated by Steps 2 and 3 above, respectively. Fusing the buses is prepared in Step 1 and carried out in Steps 4 ($G_{ij} =$ True if and only if buses i and j are fused together directly by a processor) and 5 ($C_{ij} = true$ if and only if buses i and j are fused together either directly by a processor of via a chain of fused buses). Writing on the buses is performed in Step 6 (each bus receives the data written to it directly by some processors) and 7 (the bus content is propagated according to the fused buses). At the end of Step 7 c_j contains the number of processors that have written on bus j (either directly or via fused buses); If c_j is one or zero, then the bus should be left alone; otherwise, a collision has happened and we place the collision marker accordingly (Step 8).

We argue that Step 4 is achievable in constant time, as follows: there is no specification of how many buses can be fused and in what combination by a DRMBM. However, a DRMBM processor should be

capable of computing the configuration of fused buses in constant time. Then the BSR processor simulating the DRMBM processor will also determine the corresponding broadcast parameters in constant time, since it is the same processor in terms of computational capabilities. Step 5 takes constant time by Proposition 2.2. The other steps are immediately achievable in constant time. ■

5 GAP, the Universality of Collision, and Real Time Considerations

As far a constant time computations are concerned, we noted a contrast between the power of conflict resolution rules for models with directed reconfigurable buses (DRMBM and DRN) on one hand, and for shared memory models (PRAM) on the other hand. According to our results, Collision is the most powerful rule on DRMBM and DRN. By contrast, we showed in Theorem 4.1 that the Combining CRCW PRAM is strictly more powerful than the Collision CRCW PRAM.

We also note that the ability of DRMBM (and DRN) to compute GAP in constant time is central to the constant time universality of the Collision rule, and also determines that exactly all DRMBM and DRN computations are in NL; we also note that GAP is NL-complete [18]. In light of this, we consider the classes $\mathbb{M}_{<GAP}$, $\mathbb{M}_{\equiv GAP}$, and $\mathbb{M}_{>GAP}$ of parallel models of computations using polynomially bounded resources (processors and, if applicable, buses), such that:

$\mathbb{M}_{<GAP}$ contains exactly all the models that cannot compute GAP in constant time, and cannot compute in constant time any problem outside NL.

$\mathbb{M}_{\equiv GAP}$ contains exactly all the models that can compute GAP in constant time, but cannot compute in constant time any problem outside NL.

$\mathbb{M}_{>GAP}$ contains exactly all the models that can compute GAP in constant time and can compute in constant time at least one problem outside NL. To our knowledge, no model has been proved to pertain to such a class.

As a direct consequence of Theorem 4.1, we can populate these three classes (or at least the first two) in a meaningful manner:

Theorem 5.1 *1.* Combining CRCW PRAM$(poly(n), O(1)) =$ BSR$(poly(n), O(1)) =$ DRMBM$(poly(n), poly(n), O(1)) =$ DRN$(poly(n), O(1)) =$ NL.

2. X CRCW PRAM$(poly(n), t(n)) \in \mathbb{M}_{<GAP}$ *for any* $X \in \{$Collision, Priority, Common$\}$.

3. Combining CRCW PRAM, BSR, DRN, DRMBM $\in \mathbb{M}_{\equiv GAP}$.

Proof. Immediate from definitions, Theorem 4.1, and Proposition 2.3. ■

We can also extend Claim 1 as follows:

Theorem 5.2 *For any models of computation M_1, M_2, and M_3 such that $M_1 \in \mathbb{M}_{<GAP}$, $M_2 \in \mathbb{M}_{\equiv GAP}$, and $M_3 \in \mathbb{M}_{<GAP}$, it holds that*

$$\text{rt-PROC}^{M_1}(poly(n)) \subseteq \mathsf{NL}/rt \quad (1)$$

$$\text{rt-PROC}^{M_2}(poly(n)) = \mathsf{NL}/rt \quad (2)$$

$$\text{rt-PROC}^{M_3}(poly(n)) \supset \mathsf{NL}/rt \quad (3)$$

Proof. Minor variations of previous arguments [8] show that those computations which can be performed in constant time on M_i, $1 \leq i \leq 3$, can be performed in the presence of however tight time constraints (and thus in real time in general). Then, Relations (1) and (3) follow immediately from Claim 1. By the same argument, rt-PROC$^{M_2}(poly(n)) \supseteq$ rt-PROC$^{\text{CRCW F-DRMBM}}(poly(n))$ holds as well. The equality (and thus Relation (2)) is given directly by the arguments that support Claim 1 [8]. ■

Thus, the characterization of real-time computations established by Claim 1 does hold in fact for any machines that are able to compute GAP in constant time. The characterization presented in Theorem 5.2 emphasizes in fact the strength of Claim 1. Indeed, as noted previously (Theorem 4.1), no model more powerful than the DRMBM is known to exist. That is, according to the current body of knowledge, $\mathbb{M}_{>GAP} = \emptyset$. Unless this relation is found to be false, Claim 1 states essentially that no problem outside NL can be solved in real time no matter the model of parallel computation being used—that is, Claim 1 holds for all the parallel models, not just the DRMBM.

6 Conclusions

We found that Collision is the most powerful on DRN and DRMBM for any running time. According to our results, the discussion regarding the practical feasibility of rules like Priority or Combining on spatially distributed resources such as buses is no longer of interest. Indeed, such rules are not only of questionable feasibility, but also not necessary! We thus offer a powerful tool in the analysis of models with directed reconfigurable buses and in the design of algorithms for these models: One can freely use Combining (Priority, etc.) rules on these models, with the knowledge that they can be eliminated without penalty—the analysis or algorithm design uses an "unfeasible" model yet is fully pertinent to the real world. In fact we used such a technique ourselves in the proof of Lemma 4.6.

By contrast with models with directed reconfigurable buses, it was widely believed that the all-powerful Combining conflict resolution rule does add computational power to the PRAM model, and that the BSR's Broadcast instruction adds further power. We are to our knowledge the first to establish formally a hierarchy of the PRAM variants that both confirms and contradicts the mentioned belief. Indeed, we showed that Combining does add computational power over "lesser" rules. However, we also showed that surprisingly enough the Broadcast instruction does not add computational power over Combining . In fact we established an intriguing collapse of the hierarchy of parallel models at the top of the food chain, where the Combining CRCW PRAM, the BSR, and the models with directed reconfigurable buses turn out to have identical computational power.

This result offers substantial support for analyses on shared memory models. Indeed, the power of BSR's Broadcast instruction has attracted attention in various areas of parallel algorithms. Algorithms with running time as fast as constant have been developed for various problems, most notably in the areas of geometric and graph computations [14, 15]. Of course, constant time algorithms for such practically meaningful problems are very attractive. Nevertheless, the model tends to be frowned upon given its apparent implementation complexity, even if very efficient implementations have been proposed [3]. In addition, we are not aware of any massively parallel machine that implements the BSR's Broadcast instruction, but many such machines implement the PRAM model [1, 23]. Our result shows that such an instruction—and thus the full power of the BSR model—can be used in algorithm design and analysis while keeping the analysis or the design practically pertinent. In effect, we now showed that whether the BSR is feasible or not is irrelevant, as a Combining PRAM with the same performance and with all the attractive properties of the BSR is automatically available. Same goes for the BSR versus the DRMBM (or DRN): one can freely use the BSR model (an abstract, powerful model) to design DRMBM algorithms (or VLSI circuits) and the

other way around—in essence, one can freely choose between a number of models, depending on no matter what issues (ranging from practical feasibility to convenience to mere taste) with the formally supported knowledge that the results are portable to all the other models.

This all being said, we note that simulating the Broadcast instruction on a Combining CRCW PRAM implies a (polynomial) blow-up in the number of processors used. This does not change complexity-theoretical results, but does have practical implications. Unfortunately, we believe that such a tight simulation of the Broadcast instruction is not possible without a blow-up in the number of processors.

We also note that most of our proofs are constructive (and those which are not still offer constructive hints), so we also set the basis for automatic conversion back and forth between models. True, we did not have efficiency in mind, so our constructions are likely to be inefficient; historically, however inefficient algorithms have always been optimized sooner or later, so we believe that our however sub-optimal algorithms are nonetheless a significant contribution (in addition to establishing the actual equivalence).

Beside the significance of our results by themselves, these results become particularly significant in conjunction with previous work on real-time parallel computations [8]. In this respect, we noted the central role of the graph accessibility problem for the DRN and DRMBM results obtained here and also previously. We further strengthened our previous results on real-time computations, eliminating to some degree their weak point (model dependence). In addition, the Combining CRCW PRAM joins the family of models that can solve all the problems known to be solvable under no matter how tight real-time constraints. This establishes the PRAM as the simplest complete model of computation for real time.

References

[1] F. ABOLHASSAN, R. DREFENSTEDT, J. KELLER, W. J. PAUL, AND D. SCBEERER, *On the physical design of PRAMs*, Computer, 36 (1993), pp. 756–762.

[2] S. G. AKL, *Parallel Computation: Models and Methods*, Prentice-Hall, Upper Saddle River, NJ, 1997.

[3] S. G. AKL AND L. FAVA LINDON, *An optimal implementation of broadcasting with selective reduction*, IEEE Transactions on Parallel and Distributed Systems, 4 (1993), pp. 256–269.

[4] S. G. AKL AND G. R. GUENTHER, *Broadcasting with selective reduction*, in Proceedings of the IFIP 11th World Congress, G. X. Ritter, ed., San Francisco, CA, 1989, North-Holland, Amsterdam, pp. 515–520.

[5] Y. BEN-ASHER, K.-J. LANGE, D. PELEG, AND A. SCHUSTER, *The complexity of reconfiguring network models*, Information and Computation, 121 (1995), pp. 41–58.

[6] Y. BEN-ASHER, D. PELEG, AND A. SCHUSTER, *The power of reconfiguration*, Journal of Parallel and Distributed Computing, 13 (1991), pp. 139–153.

[7] S. D. BRUDA, *The graph accessibility problem and the universality of the collision CRCW conflict resolution rule*, WSEAS Transactions on Computers, 10 (2006), pp. 2380–2387.

[8] S. D. BRUDA AND S. G. AKL, *Size matters: Logarithmic space is real time*, International Journal of Computers and Applications, 29 (2007), pp. 327–336.

[9] M. FURST, J. B. SAXE, AND M. SIPSER, *Parity, circuits, and the polynomial-time hierarchy*, Mathematical Systems Theory, 17 (1984), pp. 13–27.

[10] R. GREENLAW, H. J. HOOVER, AND W. L. RUZO, *Limits to Parallel Computation: P-Completeness Theory*, Oxford University Press, New York, NY, 1995.

[11] T. HAGERUP AND T. RADZIK, *Every robust CRCW PRAM can efficiently simulate a PRIORITY PRAM*, in Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 1990, pp. 117–124.

[12] J. JAJA, *An Introduction to Parallel Algorithms*, Addison Wesley, 1992.

[13] R. MILLER, V. L. PRASANNA-KUMAR, D. I. REISIS, AND Q. F. STOUT, *Parallel computations on reconfigurable meshes*, IEEE Transactions on Computers, 42 (1993), pp. 678–692.

[14] J.-F. MYOUPO AND D. SEME, *Work-efficient BSR-based parallel algorithms for some fundamental problems in graph theory*, The Journal of Supercomputing, 38 (2006), pp. 83–107.

[15] J.-F. MYOUPO, D. SEME, AND I. STOJMENOVIC, *Optimal BSR solutions to several convex polygon problems*, The Journal of Supercomputing, 21 (2002), pp. 77–90.

[16] I. PARBERRY, *Parallel Complexity Theory*, John Wiley & Sons, New York, NY, 1987.

[17] L. STOCKMEYER AND U. VISHKIN, *Simulation of parallel access machines by circuits*, SIAM Journal on Computing, 13 (1984), pp. 409–422.

[18] A. SZEPIETOWSKI, *Turing Machines with Sublogarithmic Space*, Springer Lecture Notes in Computer Science 843, 1994.

[19] J. L. TRAHAN, R. VAIDYANATHAN, AND R. K. THIRUCHELVAN, *On the power of segmenting and fusing buses*, Journal of Parallel and Distributed Computing, 34 (1996), pp. 82–94.

[20] R. VAIDYANATHAN AND J. L. TRAHAN, *Dynamic Reconfiguration: Architectures and Algorithms*, Springer, 2004.

[21] B.-F. WANG AND G.-H. CHEN, *Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems*, IEEE Transactions on Parallel and Distributed Systems, 1 (1990), pp. 500–507.

[22] ——, *Two-dimensional processor array with a reconfigurable bus system is at least as powerful as CRCW model*, Information Processing Letters, 36 (1990), pp. 31–36.

[23] X. WEN AND U. VISHKIN, *PRAM-on-chip: first commitment to silicon*, in Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 2007, pp. 301–302.