A Case for Reconfigurable Parallel Architectures for Information Retrieval

W. A. Woods, H. D. Moser, and O. Frieder
Department of Computer Science
George Mason University
Fairfax, VA 22030-4444
{woodswa, hmoser, ophir}@cs.gmu.edu

Paul B. Kantor
RUTCOR and SCILS,
Rutgers University
4 Huntington St., New Brunswick, NJ 08904
kantor@zodiac.rutgers.edu

Abstract

As the volume of data and computational requirements of modern information retrieval systems continue to expand, it is inevitable that parallel systems will be necessary to meet these demands. In this research, we provide a conceptual model of a reconfigurable parallel information retrieval system in a multicomputer environment. We develop strategies for scheduling queries in such systems, provide simulation results for implementing these strategies under various different theoretical situations, and present an analytical model of the system behavior.

1 Introduction

The complexity and size of modern problems continues to challenge the limits of even the most powerful serial computers. This fact dictates the need for parallel solutions to computational problems in almost every discipline. The field of information retrieval is one such domain. Contemporary databases and knowledgebases often contain gigabytes, even terabytes, of information. Searching such databases using traditional sequential machines is at best impractical and certainly will not provide satisfactory performance for interactive systems. Multicomputer environments can potentially provide the required performance for such information retrieval systems. Further, reconfigurable parallel architectures which offer the capability to dynamically group available processors can most effectively match processing power to information retrieval requirements.

Exploiting these capabilities to the fullest extent is neither simple nor straightforward. This particular work is part of a larger effort to develop a RIME (Retrieval of Information from a Multiprocessor Engine) prototype which will provide superior information retrieval performance in a reconfigurable, multicomputer environment. Specifically, we are concerned with developing optimal reconfiguration/query scheduling strategies for parallel information retrieval systems on a hypercube multicomputer.

We consider a problem which arises as reconfigurable parallel architectures are applied to parallel information retrieval [ba:91, ba:92, fr:91] where the volume of queries will make it necessary to optimize the reconfiguration strategy. Possible solutions to parallel and distributed problems of this nature are discussed in [fr:94, ki:94, se:94]. In this effort, we consider a more explicit notion of reconfiguring available processors to match the demands of each job. We design and test, via simulation, several reconfiguration strategies which can be used for efficient processing in such systems. We also discuss possible approaches to the deeper underlying problem. We presume that there are broad classes of queries for which one particular architecture is better than another. We show that failure to change the interconnection structure of an architecture from time to time may result in unbounded queues, which can be reduced or eliminated by a sensible reconfiguration policy. Finally, we address the question of how a system might learn to classify the queries with regard to architecture. Such a classification is presumed known in the present discussion.

The remainder of this paper is organized as follows: Section 2 identifies our research hypothesis. Section 3 is an overview of parallel reconfigurable information retrieval systems and section 4 provides a background of reconfigurable systems in general. Section 5 introduces the computational requirements of boolean retrieval as they relate to our problem. In sections 6 and 7, we describe the simulation and discuss results under a variety of different situations. In section 8, we develop a analytical model of how the system should behave and contrast this theoretical model with the simulation results. Section 9 explains our future plans and research extensions to this work.

2 Research Hypothesis

We propose that the relationship between the query arrival rate and the query processing rate in a parallel information retrieval system results in four possible cases. In the first case, the demand on the system is very low relative to the processing rate and both switching and non-switching strategies provide acceptable performance. At the opposite extreme, the query arrival rate could saturate the system such that the queues of waiting queries grow without bound and neither strategy provides acceptable performance. Further, we acknowledge that there exist situations in which the overhead of reconfiguration is not justified and non-switching strategies will provide superior performance. Finally, our efforts in this project are motivated by the hypothesis that there exist certain system load conditions in which simple, heuristic, reconfiguration strategies will yield acceptable performance while non-switching strategies will result in system saturation and unbounded queue lengths. Our purpose is to identify the conditions that result in this situation and to develop and test reconfiguration strategies for peak system performance in such cases. Further, we intend to develop an analytical model that will suggest optimal (possibly non-unique) switching strategies under particular operating conditions.

3 Parallel Reconfigurable Information Retrieval Systems

Information retrieval systems typically consisting of three levels: the user interface (level 1), the query manager (level 2), and the parallel processors (level 3). In this work, we are primarily concerned with developing techniques for optimizing the activities completed at Level 2. At this level, the system must be capable of identifying and reacting to queries received from the users. Similar documents in a parallel information retrieval system are typically retrieved as a group from secondary storage. These groupings can vary greatly in size resulting in increased computational requirements. Different collections are required (relevant) for different queries. Additionally, more complex queries often contain significantly more terms or involve more intricate boolean comparisons. As the number of terms and comparisons increase, so do the computational requirements for solving the query. Consequently, scheduling a query for efficient processing can require system reconfiguration to devote the proper processing power to the job and avoid a significant processing penalty.

Reconfigurable architectures can potentially offer greatly increased capabilities and complexities compared to traditional multiprocessor systems. Innovative design, analysis, and implementation techniques are required to deal with these complexities and to benefit from these capabilities. We now provide the necessary background information and general definitions assumed throughout the remainder of this paper.

4 Reconfigurable Systems

4.1 Background

Reconfiguration is generally used in systems for two reasons: (1) to map a problem to an architecture [ya:85, si:92] and (2) to provide fault tolerance [le:87, li:91]. We consider a Reconfigurable Parallel Information Retrieval System consisting of several processors with access to a common data store and an interconnection network which can be configured, via software, to map a problem to an architecture. The system receives and processes broad classes of queries to the database. We define class so that it corresponds to running much better on one configuration than another. Therefore, one particular architecture will provide superior performance for each different class of queries. This situation is similar to one which is often found in publications concerned with solving the "job shop" problem [ew:90, fi:92, we:92]. Various reconfiguration strategies can be found in literature dealing with queuing systems and simulations [ba:75, kl:75, kn:92]. Some possible options are: never reconfigure, reconfigure to match every query received, or queue queries that do not match the current architecture and reconfigure the system, based on some switching rule, to process queries waiting in the queue. Our main interest is the latter situation, and we develop simple switching rules to provide improved query processing performance

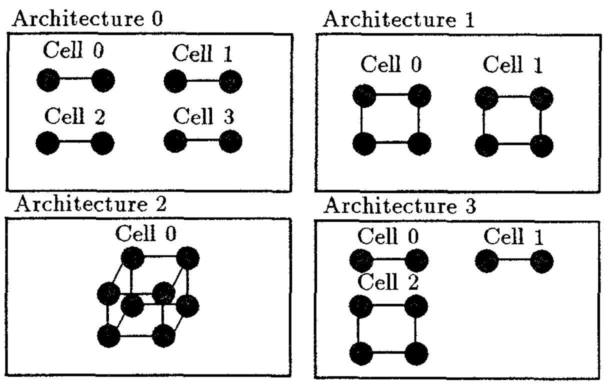

Figure 1: Simulation Architecture.

based on different operating conditions. Simple switching rules are advocated since the processing overhead that they impose is relatively low compared to more elaborate strategies. Further, we provide an analytical model of the system that can be used to "tune"our switching rules to optimize performance.

4.2 Simulation Architecture

To test possible switching strategies, we developed a simulated Reconfigurable Parallel Information Retrieval System, based on the model introduced in [wo:94], which can be configured into four possible architectures. We assume that three classes of queries can be received by the system. These classes are broadly defined as simple, moderate, and complex respectively, and we assume that the class of a query is known when it arrives at the system. We base our simulated architecture on a hypercube multicomputer with eight nodes (processors) each having separate memory and access to a common data store. A simple model is used to allow us to focus on the effects of the switching paradigm rather than on the architecture aspects. In the hypercube environment, the available nodes can be configured into "cubes" of size 1 to P where P is the number of available processors ($P = 8$ in our simulation). We define four possible architectures based on different configurations of the eight available processors. Figure 1 shows these architectures. Notice that each architecture consists of one or more cubes or cells. If the size of the cell matches the complexity of the query, we assume the query can be efficiently processed by that particular cell. As mentioned earlier, we define three classes of queries that we expect will be received by the system: Type 0 - simple, Type 1 - moderate, and Type 2 - complex. We assume that the simple queries require little processing power and can be efficiently satisfied using a cell with 2 nodes. A complex query requires a cell containing all 8 nodes, and a moderate query can be efficiently processed on a cell containing 4 nodes.

5 Boolean Systems as an Example

Boolean retrieval represents a well-known specific example of the general formulation. The evaluation function $\pi(q, d)$ is defined by applying the compound characteristic function defined by the terms in the query, to the set of terms used to represent the document. Thus, if the term combination

$t1$ AND $t2$ appears in the query, the corresponding characteristic functions are multiplied to define π.

$$\pi(q,d) = \chi_{t_1}(d) \times \chi_{t_2}(d) \quad (1)$$

The ordered set L is simply the set $\{0,1\}$, with the total order $1 > 0$. The extension to fuzzy sets replaces the product by the minimum, and the ordered set L becomes the unit interval [0,1]. In this situation it is known that the most efficient computation is by means of an "inverted file". This is a data structure which constructs, for each term t not a stop word, a file of documents $D_t = \{d : t(d) = 1\}$. In a parallel computation scheme, these several files will be distributed over the nodes of the system. The basic computational operations conducted on these files are the formation of the intersection and the union. If all the files are sorted, which is clearly optimal, both of these computations have complexity $O(m+n)$, where m is the length of the longer file, and n is the length of the shorter file. But the coefficients, in determining the number of compute cycles, or the total time required for the computation, depend on where the files are located. Let c_s be the time of the unit comparison and move, when the files are located at the same node. Then, for both files at the same node, the time is

$$t_s(m,n) = c_s(m+n) \quad (2)$$

If the two files are located at different nodes, then the cost is increased by a communication term, which is, optimally, applied only to the shorter file. Let l represent the link structure which is established between the two nodes, and let c_l represent the time required to move a single file element across those links. Then the time for comparison when the files are not on the same node is

$$t_l(m,n) = t_s(m,n) + c_l n \quad (3)$$

Clearly this is greater than t_s. The situation described relates to a distributed database but is directly analogous to a common data store in which multiple clusters are accessed in a complex query.

The terms which appear in a query will, therefore, determine how long it takes to process. If all of the terms have files located at the same node, the processing will be quicker than if they are not. When there are several nodes, the architecture can be varied, to change the link path between particular pairs of nodes. Thus, if the files relevant to a particular query are divided between two nodes, the retrieval will be quickest when the link between those two nodes is as short as possible. Once that architecture is established, it makes sense to process all the pending queries for which it is optimal, and to do so with new arrivals until the backlog of deferred queries becomes too large.

6 A Multi-Class Simulation

6.1 General Situation

We define a parallel information retrieval system which has four possible configurations. Depending on the type of query received, one configuration will be superior over the other in terms of processing time per query. In particular, type 0 queries can be best processed when the system is in architecture 0 or architecture 3. Type 1 queries can be best processed in architecture 1 or architecture 3, and type 2 queries can be most efficiently processed only in architecture 2. The processing demand and data accessed by a particular type of query directly determines the degree of match between that type of query and a particular architecture. For example, we state that type 1 queries can be best processed in architecture 1 or architecture 3 since each of these architectures contain cells with 4 nodes (a type 1 query is most efficiently processed using 4 nodes). Upon receipt of a query, the system can immediately determine the type of the query based on accessed data volume and stated resource demands. Consequently, the best configuration for processing the query is also known. The disposition of an incoming query depends upon the type of the query and the current system configuration. If the query type matches the configuration of the system and the type of the next available cell, it is scheduled for processing, otherwise, it is placed in a queue.

6.2 System Reconfiguration Strategy

The decision to reconfigure the system and process queued queries can be determined by a variety of switching strategies. We present results in this paper based on three different switching strategies.

- *Rule a.* If the queue of queries not matching the current architecture exceeds some threshold, then reconfigure the system and process some portion (possibly all) of the waiting tasks. The motivation for this rule is to ensure that queue lengths and job processing times are controlled.
- *Rule b.* If a type 2 query arrives, then switch to architecture 2 and process that arrival immediately (non-preemptive). The motivation for this rule is to model a situation where critical or high priority queries exist. In such an environment, our objective would be to limit, or eliminate, the delay experienced by these high priority queries.
- *Rule c.* If a queue contains a waiting query that has experienced an unacceptable delay penalty, then reconfigure the system and process some portion (possibly all) of the waiting tasks. The delay penalty is computed as a function of time t that the query has been in the queue (e.g., $2t$, t^2, etc.)

To judge the benefit of system reconfiguration, we provide results in which dynamic reconfiguration is not possible. In this case, the system is "locked" in one of the four possible architectures. In the non-switching condition, we assume that all queries are processed on a first-come-first-served basis. While those queries that do not match the architecture can be satisfied, if the size of the processing cell is smaller than required, processing will be slower.

6.3 Arrival Streams

We assume that the arrival rate of queries to the system follows a poisson distribution. We generated numerous arrival streams representing different system load conditions. Using these arrival streams, we conducted simulations under three general situations: (1) All types of queries arrive at the same rate. This can be represented by the triplet $\langle x, x, x\rangle$, where each value in the triplet represents the arrival rate for each type of query; (2) Simple queries arrive very infrequently $\langle 10.0, x, x\rangle$, where $10.0 \gg x$; and (3) moderate and complex queries arrive very infrequently $\langle x, 10.0, 10.0\rangle$, where again $10.0 \gg x$.

6.4 Simulation Parameters and Measures of Performance

Query processing times are also assumed to be exponentially distributed. However, the mean processing time depends on both the system configuration and the type of query being processed. In general, we assume linear speedup such that a query which requires 4 nodes (moderate) would require twice as long to process on 2 nodes. Naturally, reconfiguration of the system is not without cost and we assume a system reconfiguration time based on the number of processors affected by the reconfiguration. Finally, we stop the simulation after a total of 2000 queries have been delayed in the queue.

We use several measures of performance for this simulated system which are described in detail by [la:91]. The primary performance measures and simulation parameters we use are:

$\hat{d}$ - expected task delay in the queue,
$\hat{q}$ - expected number of tasks in the queue, and
$\hat{u}$ - expected proportion of time the system is busy.

The derivation of these measures is shown below:

$$\hat{d} = \frac{1}{n}\sum_{1}^{n} d_i \tag{4}$$

where d_i is the delay in queue of the ith task.

$$\hat{q} = \frac{1}{t_{stop}}\int_0^{t_{stop}} q_t \, dt \tag{5}$$

where q_t is the queue length at time t.

$$\hat{u} = \frac{1}{t_{stop}}\int_0^{t_{stop}} b_t \, dt \tag{6}$$

where

$$b_t = \begin{cases} 1 & \text{if system is busy} \\ 0 & \text{if system is not busy} \end{cases}$$

7 Simulation Results

As in [wo:94], we present and discuss the results of simulations based on different operating conditions and different switching strategies. In this effort, we provide experimental results using additional, more varied reconfiguration rules. Each sub-section that follows describes the situation modeled and the simulation results. In each case, we use the same set of rules to determine when to reconfigure the system. Specifically, we test the three switching strategies described above as *Rule a*, *Rule b*, and *Rule c*. For *Rule a* and *Rule c* we used a value of 10 as the threshold to trigger reconfiguration. For *Rule b*, reconfiguration is controlled by the arrival of a type 2 query regardless of the state of any other system variables. We apply these switching rules under three different operating conditions: (1) all types of jobs arrive at the same rate; (2) simple jobs arrive very infrequently; and (3) moderate and complex jobs arrive very infrequently. Under each proposed operating condition, we test arrival streams which represent very low to very high processing demands on the system. As a measure of the value of system reconfiguration, we compare the system performance when switching is used to the system performance when the system is locked in one of the four architectures and no reconfiguration is possible. We use the average task delay in queue as our measure of performance for this comparison.

Table 1 shows the relative performance of the switching strategies compared to the non-switching mode under various levels of processing demand. The table is divided into three sections (A, B, and C) where each section corresponds to one of the three operating conditions mentioned above. Each row of the table represents a particular system processing demand. This is identified by an arrival rate triplet which lists the arrival rate for type 0, type 1, and type 2 queries respectively. Each cell in the table shows the average query delay in queue when switching is used compared to the average query delay in queue when no switching is used ($d_{switching}/d_{non-switching}$). For example, the first column in the table lists the relative performance of switching *Rule a* compared to non-switching when the system is locked in architecture 0, the second column lists the same results when the non-switching architecture is 3, etc. Note that four columns are displayed for each switching rule and that the non- switching architecture columns are listed in order 0, 3, 1, 2 in order to highlight situations when switching is most beneficial. This is shown by the shaded cells which have a value less than 1.0.

7.1 All Types Arrive at the Same Rate

Table 1A shows that switching rules a and c can often provide better performance during periods of high demand on the system. Note that switching *Rule b* generally yields poor performance under these operating conditions. This observation is logical since all types of queries arrive at the same rate and each time a type 2 query arrives, the system must reconfigure to architecture 2 in order to process the query.

7.2 Simple Tasks Arrive Infrequently

In this situation, the arrival rate of simple queries is much lower compared to moderate and complex queries. In Table 1B, we show that under these conditions a non-switching strategy typically provides equal or superior performance compared to any of the switching rules. *Rule a* does perform slightly better under peak system load conditions which indicates that the overhead of switching may be justified when demand on the system is greatest. Otherwise, switching is only beneficial when compared to non-switching if the system is locked in architecture 2. Clearly, this would be a poor choice unless almost all of the tasks were complex since in architecture 2 only one cell is available for processing tasks and considerable computational power is wasted when processing simple or moderate queries.

7.3 Moderate and Complex Tasks Arrive Infrequently

In this situation, simple queries are much more frequent than moderate or complex queries. An analogous situation could be an environment where routine queries were very frequent but critical or high priority queries requiring immediate response and considerable processing power were relatively rare. We propose that this is possibly the most realistic situation that may be observed in an actual information retrieval system. In Table 1C, we show that at high system load conditions any of the switching strategies tends to provide better performance.

		Rule a				Rule b				Rule c			
		Non-Switching Architecture				Non-Switching Architecture				Non-Switching Architecture			
	Arrival Rate Triplet	0	3	1	2	0	3	1	2	0	3	1	2
A.	**All types arrive at the same rate $\langle x, x, x\rangle$**												
High	⟨0.57,0.57,0.57⟩	1.27	0 87	0 64	0.22	2.81	1.93	1.42	0.48	1.37	0.94	0.69	0.23
Processing	⟨0.53,0.53,0.53⟩	2.04	0.85	0.50	0 12	7.11	2.97	1.74	0.41	0.93	0.39	0.23	0.05
Demand	⟨0.49,0.49,0.49⟩	2.87	1.12	0.52	0.07	12.82	5.00	1.89	0 30	3.49	1 36	0.51	0 08
Low	⟨0.46,0 46,0.46⟩	4.88	2.08	0.70	0.05	22.65	9.65	3.23	0.24	3.80	1.62	0 54	0.04
Processing	⟨0.43,0.43,0.43⟩	6.75	4.01	2.13	0.05	13.21	7.85	4.16	0 09	5 99	3.56	1.89	0 04
Demand	⟨0.40,0 40,0.40⟩	9.31	5.89	3.20	0.05	9.71	6.14	3 33	0.05	7.48	4.73	2.57	0.04
B.	**Simple tasks arrive infrequently $\langle 0.10, x, x\rangle$**												
High	⟨0.10,0.69,0.69⟩	1.06	0.99	0.94	0.35	1.40	1.31	1.24	0 47	1.49	1 40	1 32	0.50
Processing	⟨0.10,0 66,0.66⟩	0 98	0.90	0.84	0 27	1.73	1.58	1.48	0.47	1.71	1.57	1.46	0 46
Demand	⟨0 10,0.64,0.64⟩	1.24	1.13	1 03	0.29	2.27	2.06	1 88	0.53	2.46	2.25	2.04	0.58
	⟨0.10,0.63,0.63⟩	1.34	1.16	1.05	0.26	2.51	2.18	1.98	0.49	1.70	1.47	1 33	0.33
Low	⟨0.10,0.61,0.61⟩	1.49	1.19	1.03	0.19	3.89	3.10	2.68	0.50	2.96	2.36	2.05	0 38
Processing	⟨0.10,0.59,0.59⟩	1.60	1.33	1.10	0.19	4.01	3.32	2.76	0.48	3 99	3 30	2 74	0.48
Demand	⟨0.10,0.58,0.58⟩	2.09	1.52	1.19	0.14	6.56	4 76	3.73	0 45	5.93	4 30	3.37	0 40
	⟨0.10,0.56,0 56⟩	1.89	1.50	1.21	0.13	6.83	5.44	4 40	0.47	4.92	3.92	3.17	0.34
C.	**Moderate and complex tasks arrive infrequently $\langle x, 0.10, 0.10\rangle$**												
High	⟨4.15,0 10,0.10⟩	0.49	0.22	0.10	0.04	1.58	0.70	0.34	0 13	0.61	0.27	0 13	0.05
Processing	⟨2.83,0.10,0.10⟩	2.34	0.17	0.03	0.02	5.24	0.37	0.02	0.04	7.25	0.51	0.17	0.05
Demand	⟨2 13,0.10,0.10⟩	10.90	0.85	0.05	0.01	7.79	0.61	0.02	0.01	81.80	6.36	0.33	0.07
Low	⟨1.69,0.10,0.10⟩	19.74	3.68	0.08	0.01	8 58	1.60	0.04	0.01	138.10	25.73	0 59	0 08
Processing	⟨1.39,0 10,0.10⟩	33 05	8 16	1.14	0.01	9.80	2.42	0.34	0.00	135.10	33.36	4.66	0.05
Demand	⟨1.17,0 10,0.10⟩	69.73	18.70	3.87	0.02	14.82	3.98	0 82	0.00	147.60	39.59	8 20	0.04

Table 1: Simulation Results.

8 Analytical Model

In an attempt to improve the capabilities of the system, an analytical system model was developed. This modeling effort utilized a linear programming approach to determine asymptotic bounds on the capabilities of the system. From this, comparisons were made to the simulation based system, and conclusions were drawn as to the switching rules that were utilized. The primary motivation for this approach was to determine (a) if the realization of the system architecture allowed for a stable service effort for the incoming query stream, and (b) if the switching strategy was pushing the system closer to a desirable balance of effort as measured by the percentage of time that the system spends processing in any particular architecture.

In developing the model, we used the fact that for the queuing system to remain stable, the traffic intensity ratio of arrival rate λ to service rate μ must be less than one. For the general case of $M/M/k$ queues, where there are exponential arrivals and service times, and k servers, this requirement is stated as $\lambda/k\mu < 1$.

Extending this notion to our situation requires introducing the following notation:

Q	=	set of query types numbered $i = 0, 1, 2$
A	=	set of architecture types numbered $j = 0, 1, 2, 3$
λ_i	=	arrival rates for query type i
μ_i	=	service rate for query type i in cell type i
$k_{i,j}$	=	number of cells of type i in architecture j
P_j	=	percentage of time that the system remains in architecture j

A necessary system requirement, therefore, is the following: the system must process all queries for a sufficient percentage of the total time such that the queues remain stable. For the case where there is only one architecture, j, that contains cells of type i, this can be interpreted as $p_j > i/(k_{i,j}\mu_i)$. In our example, queries of type 2 fall into this category, which leads to the requirement that:

$$p_2 > \frac{\lambda_2}{\mu_2} \tag{7}$$

where $k_{2,2} = 1$. For query types 0 and 1, the procedure for defining the percentages requires additional effort. Beginning with type 1, let $\lambda 1/(p_1 k_{1,1} \mu 1)$, be the service contribution of architecture 1 to servicing queries of type 1. Similarly, $\lambda 1/(p_3 k_{1,3} \mu_1)$ is the service contribution of architecture 3 to query type 1. To combine the effects of the two architecture types on overall system stability, we compute the following relation (where $k_{1,1} = 2$ and $k_{1,3} = 1$):

$$\frac{1}{\frac{1}{\frac{\lambda_1}{2p_1\mu_1}} + \frac{1}{\frac{\lambda_1}{p_3\mu_1}}} < 1 \tag{8}$$

which yields:

$$2p_1 + p_3 > \frac{\lambda_1}{\mu_1} \tag{9}$$

A similar relationship for query type 0 can be derived as

$$4p_0 + 2p_3 > \frac{\lambda_0}{\mu_0} \tag{10}$$

In general, then we can state that there will be a constraint relationship derived for each element of Q, and there will be a decision variable, p_j, corresponding to each element of A.

As a tool to derive a bound on system performance, therefore, we developed a linear program (LP) that seeks to determine the minimum percentage of time that the system must spend processing queries in order to remain stable, subject to the stability requirements given above, and the feasibility requirement that the sum of the p_j be less than or equal to 1. The general model is given as:

$$\begin{array}{l} \text{Linear Programming Model (LP)} \\ \text{minimize } p^* = \sum_{j \in A} p_j \\ \text{subject to:} \\ \sum_{j \in A} k_{i,j} p_j > \frac{\lambda_i}{\mu_i}, \forall i \in Q \\ \sum_{j \in A} p_j \leq 1 \\ p_j \geq 1, \forall j \in A \end{array} \tag{11}$$

The specific formulation for our example, then, is given as LP1 below:

$$\begin{array}{l} \text{Linear Programming Model (LP1)} \\ \text{minimize } p^* = \sum_{j=0}^{2} p_j \\ \text{subject to:} \\ 4p_0 + 2p_3 > \frac{\lambda_0}{\mu_0} \\ 2p_1 + p_3 > \frac{\lambda_1}{\mu_1} \\ p_2 > \frac{\lambda_2}{\mu_2} \\ \sum_{j=0}^{2} p_j \leq 1 \\ p_j \geq 0, j = 0, 1, 2 \end{array} \tag{12}$$

Here, we note that in solving LP1 for the minimal overall percentage of processing time required for the system to remain stable, we compute an instance of individual values of p_j^* (not necessarily unique) that yields p^*. These p_j^* then, are an instance of the set of all possible p_j that would allow for the queues to remain just barely stable.

With this information, we compared the results of LP1 to the values of p_j that were observed during the simulation runs. This information provides insight into possible modifications to our reconfiguration strategies to improve the performance of the system parameters.

9 Conclusions and Future Research

The results presented here are encouraging. It appears that proper switching strategies, intelligently implemented, can improve the performance of Reconfigurable Parallel Information Retrieval Systems. However, additional effort is required to apply these results to an actual system. In particular, data from an operational information retrieval system is necessary to fix the form of the distributions used in our simulation.

Our effort models an eight node system with only three classes of queries. While this provides some insight into the stated problem, it is clearly not sufficient to develop techniques for implementation in realistic information retrieval systems. Additionally, our results, thus far, are confined to only three reconfiguration strategies based on fairly restrictive assumptions. We will enhance the current simulation to model a greater number of possible configuration types and query classes to represent realistic systems. Further, we will conduct experiments with a wide variety of incoming query types, with varying system parameters, and develop a combination of heuristic and analytical reconfiguration strategies for efficiently processing such arrival streams. Finally, we will use the optimal steady state percentages predicted by our analytical model to "tune" our switching rules which should further improve our system performance parameters.

Acknowledgments

This work was supported in part by the National Science Foundation under contract #IRI-9357785, the Virginia Center for Innovative Technology under contract #INF-94-002 and by the XPAND Corporation. The authors wish to thank Mark Maloof for his assistance with the final preparations of this manuscript.

References

[ba:91] M. Baig, T. El-Ghazawi, N. Alexandridis. *A Highly Reconfigurable MSIMD/MIMD Architecture.* In Proc. 4th ISMM PDCS, Washington, D.C., 1991.

[ba:92] M. Baig, T. El-Ghazawi, and N. Alexandridis. *Mixed-Mode Multicomputers with Load Adaptability.* In Parallel Architecture and Languages, IEEE CS, 1992.

[ba:75] F. Baskett. *Open, Closed, and Mixed Networks of Queues with Different Classes of Customers.* JACM, 22(2), 1975, pp. 231–247.

[ew:90] K. Ewacha, I. Rival, G. Steiner. *Permutation Schedules for Flow Shops with Precedence Constraints.* OR 38(6), 1990, pp. 1135–1139.

[fi:92] M. Fischetti, S. Martello, P. Toth. *Approximation Algorithms for Fixed Job Schedule Problems.* OR, 40(supp. 1), 1992, pp. S96–S108.

[fr:91] O. Frieder, H. T. Siegelmann. *On the Allocation of Documents in Multiprocessor Information Retrieval Systems.* ACM SIGIR, 1991, pp. 230–239.

[fr:94] O. Frieder, C. K. Baur. *Site and Query Scheduling Policies in Multicomputer Database Systems.* IEEE TKDE, 6(4), 1994, pp. 609–619.

[kl:75] L. Kleinrock. *Queueing Systems, Volume 1: Theory.* John Wiley & Sons, 1975.

[kn:92] C. Knessl, C. Tier. *A Processor-Shared Queue that Models Switching Times: Normal Usage.* SIAM Journal on Applied Mathematics, 52(3), 1992, pp. 883–899.

[ki:94] T. Kindberg, A. V. Sahiner, Y. Paker. *Adaptive Parallelism under Equus.* In Proc. 2nd Workshop on Configurable Distributed Systems, IEEE CS Press, 1994, pp. 172–182.

[la:91] A. Law, W. D. Kelton. *Simulation Modeling & Analysis*, 2nd ed. McGraw-Hill, 1991.

[le:87] Y. Lee, K. G. Shin. *Optimal Reconfiguration Strategy for a Degradable Multimodule Computing System.* JACM, 34(2), 1987, pp. 326–348.

[li:91] H. Li, Q. F. Stout. *Reconfigurable SIMD Massively Parallel Computers.* Proc. IEEE, 79(4), 1991, pp. 429–443.

[se:94] S. K. Setia, M. S. Squillante, S. K. Tripathi. *Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems.* IEEE TPDCS, 5(4), 1994, pp. 401–420.

[si:92] H. J. Siegel, J. B. Armstrong, D. W. Watson. *Mapping Computer-Vision-Related Tasks onto Reconfigurable Parallel-Processing Systems.* IEEE Computer, 25(2), 1992, pp. 54–63.

[we:92] L. M. Wein *Dynamic Scheduling of a Multiclass Make-to-Stock Queue.* OR, 40(4), 1992, pp. 724–735.

[wo:94] W. A. Woods, H. D. Moser, O. Frieder, P. B. Kantor. *An Experimental Evaluation of Task Scheduling on Reconfigurable Multicomputer Architectures.* IEEE 7th PDCS, Las Vegas, Nevada, 1994.

[ya:85] S. Yalamanchili, J. K. Aggarwal. *Reconfigurable Strategies for Parallel Architectures.* IEEE Computer, 18(12), 1985, pp. 44–61.