# Purdue University Purdue e-Pubs

**ECE Technical Reports** 

**Electrical and Computer Engineering** 

4-1-1993

# On a Quantitative Model of Dynamic System Reconfiguration Due to a Fault

Gene Saghi Purdue University School of Electrical Engineering

Howard Jay Siegel Purdue University School of Electrical Engineering

Jose A. B. Fortes Purdue University School of Electrical Engineering

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

Saghi, Gene; Siegel, Howard Jay; and Fortes, Jose A. B., "On a Quantitative Model of Dynamic System Reconfiguration Due to a Fault" (1993). *ECE Technical Reports*. Paper 228. http://docs.lib.purdue.edu/ecetr/228

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

# ON A QUANTITATIVE MODEL OF DYNAMIC SYSTEM RECONFIGURATION DUE TO A FAULT

GENE SAGHI HOWARD JAY SIEGEL JOSE A. B. FORTES

TR-EE 93-18 April 1993



School of Electrical Engineering Purdue University West Lafayette, Indiana 47907-1285

# On a Quantitative Model of Dynamic System Reconfiguration Due to a Fault

Gene Saghi

Howard Jay Siegel

Jose A. B. Fortes

Parallel Processing Laboratory School of Electrical Engineering Purdue University West Lafayette, IN 47907-1285 USA

April 1993

This research was supported by the National Science Foundation under grant number CDA-9015696, by the Office of Naval Research under grant number N00014-90-J-1483, and by the Innovative Science and Technology Office of the Strategic Defense Organization and administered through the Office of Naval Research under contract number N00014-88-K-0723.

## Abstract

The use of dynamic reconfiguration has been proposed to tolerate faults in large-scale partitionable parallel processing systems. If a processor develops a permanent fault during the execution of a task on a submachine A, three recovery options are migration of the task to another submachine, task migration to a subdivision of A, and redistribution of the task among the fault-free processors in A. Quantitative models of these reconfiguration schemes are developed to consider what information is needed to make a choice among these methods for a practical implementation. It is pointed out that in certain situations collecting precise values for all needed parameters is very difficult. Therefore, the model parameters are then analyzed, together with the cost of making the wrong reconfiguration choice, to determine a useful heuristic that is based on the **information** available. A multistage cube or hypercube inter-processor network is assumed. PASM, an experimental **SIMD/MIMD** mixed-mode machine with a partitionable multistage cube communication network, and **nCUBE** 2, a commercially available MIMD machine with a partitionable hypercube communication network, are used as vehicles for studying the model parameters.

#### 1. Introduction

To provide reliable operation over extended periods of time, massively parallel processing systems must be capable of tolerating faults. A fault-tolerant system must be able to detect and locate faults, to reconfigure itself to "disconnect" and perhaps replace faulty components, to recover from possibly erroneous computations, and to restart operation from a correct state. When more than one reconfiguration option is available, the option that results in the earliest completion of the task is desirable. Quantitative models of three different reconfiguration schemes are developed in this chapter to consider what information is needed to make a choice among these methods for a practical implementation. In certain situations collecting precise values for all needed parameters is very difficult (if not impossible). Therefore, the ranges of values that the model parameters can assume are analyzed to develop guidelines for making the best reconfiguration choice. Because there is no guarantee that these guidelines will produce the optimal reconfiguration strategy in all cases, the cost penalty of making the wrong choice is also considered. The analysis incorporates experimentally-derived parameters obtained on PASM [FiC91, SiS87], an experimental **SIMD/MIMD** mixed-mode machine with a partitionable multistage cube communication network, and on nCUBE 2 [Ncu90], a commercially available MIMD machine with a partitionable hypercube communication network.

One approach to achieving fault tolerance in parallel processing systems involves the use of redundant hardware. When a faulty component is detected, the system is reconfigured in such a way that the faulty component is replaced by one of the redundant components. An example of such a system is MPP [Bat82]. MPP is a  $128 \times 128$  array of single bit processors implemented as a  $64 \times 32$  array of 2 row  $\times 4$  column VLSI ICs. A redundant column of VLSI ICs (64 ICs) is provided to replace any of the 32 columns of ICs within which a faulty processor may exist. Thus, any single processor IC failure in the array can be tolerated. Additional hardware could be added, at additional cost, to allow a greater number of faults to be tolerated. A general disadvantage of the redundant hardware approach is that the extra hardware is idle until a fault occurs.

The work presented in this chapter focuses on partitionable parallel processing systems where the set of processors can be partitioned to form multiple independent <u>submachines</u>. The execution of a parallel program on a submachine is defined as a <u>task</u>. It is possible to achieve fault tolerance in such a system by utilizing the reconfigurability of the system to effectively ''disconnect'' the faulty component. For example, parallel processing systems such as Intel Cube [Int85], nCUBE [HaM86], IBM RP3 [PfB85], and PASM [SiS87, FiC91] incorporate partitionable interconnection networks and therefore have the ability to migrate a

task from a faulty submachine to a fault-free submachine [e.g., ScS90]. Such an approach has the advantage of no redundant hardware costs in the absence of faults. However, the total available system resources are decreased when a fault occurs. Furthermore, if the smallest possible submachine in such a system consists of more than one processor, the fault-free processors in the faulty submachine become idle. Thus, there is a trade-off between redundant hardware cost and degraded system performance for the fault tolerance schemes discussed.

The architecture assumed here implements a physically distributed memory such that each processor is paired with local memory to form a processing element (PE). Most existing large-scale parallel processing systems use a physically distributed memory approach (e.g., BBN Butterfly [CrG85], Connection Machine CM-2 [TuR88], Intel Cube [Int85], nCUBE [HaM86], DAP [Hun89], MasPar [Bla90], IBM RP3 [PfB85]). These systems implement either a logically nonshared memory system, a logically shared memory system, or a hybrid of the two memory systems. In a logically nonshared memory system (e.g., CM-2, MasPar), processors cannot access remote memory locations directly. Instead, all communication between PEs is through explicit message passing. In a logically shared memory system (e.g., BBN Butterfly), all system memory appears in the address space of each processor. Accesses to memory locations located in a remote processor's memory requires use of the interconnection network. As a result, remote memory accesses incur a larger latency than local memory references. Careful placement of program code and data is one way to reduce the effects of this network latency. In one type of hybrid memory system (e.g., IBM RP3), a portion of each processor's memory is reserved for nonshared access, the remainder is treated as logically shared memory, and all interprocessor communication is through the shared memory. For the model of reconfiguration presented in this chapter, a logically shared or hybrid memory system is assumed.

Previous research on fault recovery by dynamic reconfiguration includes [UyR85] and [UyR88]. These papers explored task redistribution in a MIMD environment to recover from a PE fault for near-neighbor-class problems. Other work has considered the reconfiguration of hypercube architectures in the event of PE failure [HaL87, LiS88]. Here, quantitative models of three different reconfiguration schemes are developed. Givn today's technology, collecting precise values for all needed parameters is very difficult in certain circumstances. Therefore, the time-cost ranges of the model parameters and of the time penalty for a suboptimal reconfiguration choice are examined and compared. The PASM and nCUBE 2 parallel machines are used as vehicles for studying the model parameters. For some parameters, the range can be restricted to a very small and precise interval. For other parameters, only very coarse, but useful, ranges can be determined. It is shown that this combination of precise

and coarse ranges can be used to determine useful guidelines for a choice among reconfiguration options.

The system and fault models used to analyze fault recovery options are described in Section 2. Section 3 presents recovery options and associated costs. One cost that must be considered for every fault recovery option is the remaining execution time of the task that was on the faulty submachine when the fault occurred. As an example of the difficulties in determining some of these costs, Sections 4 and 5 provide models for determining the remaining execution time of a task depending on the system configuration and operation modes. In Section 4, tasks whose execution times are data independent are considered, while in Section 5 tasks whose execution times are data dependent are considered. An analysis of the range of costs for each option is examined in Section 6 to determine the relative weight of these costs in the reconfiguration decision. Finally, the penalty for making the wrong choice is considered in Section 8.

#### 2. System Model for Fault Recovery

The analyses here can be used to model MIMD, multiple-SIMD, or partitionable **SIMD/MIMD** parallel processing systems, utilizing a multistage cube or hypercube interconnection network, and possessing a logically shared or hybrid memory system. The research assumes a partitionable **SIMD/MIMD** machine with a multistage cube network and can be directly applied to the other cases.

It is assumed that overall system activities are supervised by a dedicated processor known as the system controller unit (SCU), although it could be a program distributed among system processors. Among other duties, the SCU is responsible for allocating and **deallocat**-ing submachines, and for determining the proper recovery action in the event of a detected fault. The activities in each submachine are supervised by a submachine controller (SC). Similar to the SCU, the SC is assumed to be a designated processor, although it could be a program distributed among the processors of the submachine.

The execution of an SIMD procedure on a submachine is an <u>SIMD process</u>. An <u>MIMD process</u> is the execution of an MIMD procedure on a PE that is part of the **sub**machine. The term <u>subtask</u> refers to a single thread of control (stream of instructions). A **subtask** can be composed of one or more SIMD processes executed sequentially on the same submachine, in which case **subtask** refers to the program executing and/or broadcast by the SIMD submachine SC. A **subtask** may also be composed of one or more MIMD processes executed sequentially on the same submachine PE, in which case **subtask** refers to the program executed by a single PE in the MIMD submachine. A task can be composed of one or more subtasks. Tasks are coded assuming that any number of *virtual processors* required is available. At execution time, the SC determines the number of physical processors available in the submachine and maps the virtual processors onto physical processors. The ratio of virtual processors to physical processors is called the <u>virtual processor ratio</u>. When the virtual processor ratio is greater than one, some or all of the physical processors will perform the functions of more than one virtual processor. If the virtual processor scheme allows tasks to be executed on submachines of various sizes without having to be re-compiled. The Connection Machine models **CM1** and **CM2** [**TuR88**] uses such a virtual processor scheme to allow programs to be executed on machines consisting of different numbers of physical processors. Other schemes that allow for the same kind of functionality are equally applicable to this research.

The model used for fault tolerance and recovery is as follows. At regular intervals during the execution of a task, the state of each PE, including register and allocated memory contents, is stored in a <u>buddy</u>, **i.e.**, a different PE within the same submachine. This state information is called <u>checkpoint data</u> and is used to restore a valid system state (i.e., a <u>recovery point</u>) in the event of a fault. No two **PEs** have the same buddy. Furthermore, buddies **are** chosen such that all **PEs** in a submachine can store checkpoint data in their buddies simultaneously without conflicts in the inter-PE communication network. The checkpoint data **are** also stored in local PE memory to allow recovery in the event its buddy becomes faulty. Thus, no matter which PE becomes faulty, a copy of that **PEs** checkpoint data is available for error-recover somewhere within the submachine. Error-recovery techniques using checkpointing are discussed in [**TaS84**] and [**FrT89**].

It is assumed that existing fault detection techniques in the literature (e.g., [DaM85] or [BaB91]) are used in the system to detect faulty components. The faults of interest are permanent faults that affect the processor of the PE or the memory module of the PE. Transient faults are not considered, because they do not require system reconfiguration. For this study it is assumed that a <u>faulty processor</u> is unable to either compute or communicate with other **PEs**, but does not interfere with the operation of fault-free **PEs**. Furthermore, the local memory of a faulty processor is assumed to be corrupt, or inaccessible. The failure of a processor local data bus or arithmetic logic unit would be examples of faults classified as processor faults. When a memory module is faulty, it is assumed that neither the local processor nor any other PE can reliably read data from or write data to the <u>faulty memory module</u>. The local processor is assumed to be fully operational in every other way. Such would be the

case if there was a failure in the memory module refresh circuitry or a memory module I/O buffer, for example. A PE with a faulty processor or a faulty memory module will henceforth be referred to as a <u>faulty PE</u>, unless it is necessary to distinguish between the two fault types. When a fault is detected, the SCU determines and directs the proper recovery action after which processing continues from the last valid checkpoint.

# 3. Quantitative Dynamic ReconfigurationModel

When a permanent fault occurs in a submachine A, four possible reconfiguration/recovery options are as follows. The first three options apply to PE faults in general, while the fourth is applicable only to PE memory module failures.

- 1) Subdivide A into two equal-size system submachines, and use the one that is fault-free to complete the execution of the task.
- 2) Migrate the task to another submachine that is fault-free.
- 3) Redismbute the task programs and data among the fault-free **PEs** in A and complete the task using a modified algorithm that does not use the faulty PE.
- 4) If a PE memory module fault occurs, using information from secondary storage or checkpoint data, load the process that was executing on that PE into the memory modules of other PEs in submachine A and continue as before, but with the processor associated with the faulty memory module accessing only remote memory.

These recovery options are discussed further in the following subsections. Table I summarizes the most important notation used throughout the sections that follow.

In addition to the costs of recovery discussed for each option, there is a time overhead of determining these recovery costs to select the best option for a given situation. However, this overhead is incurred prior to the initiation of any of these recovery schemes, so it is separate from the cost of recovery and is not included in the following subsections.

### 3.1. Task Completion on a Fault-Free Subdivision

When a PE fault occurs on a dynamically partitionable system, it can be avoided by subdividing the current submachine and completing the task on the fault-free subdivision. Figure 1 illustrates this process. It depicts a task executing on a submachine originally consisting of eight **PEs**. When PE 2 develops a fault, the task is moved to the fault-free subdivision (four **PEs**) of the original submachine, and task execution is completed there.

The fault-free subdivision recovery process would proceed as follows. Once the PE

**Table I:**Summary of notation used throughout the chapter.

Notation	Meaning	
FFS	fault-free subdivision reconfiguration option	
ТМ	task migration reconfiguration option	
TR	task redistribution reconfiguration option	
T <sup>FFS</sup> TCheck	time to select fault-free subdivision	
$T_{Map}^{YYY}$	time to determine new mapping of task for <b>YYY</b> *	
T <sup>FFS</sup> Part	time to update system partition table for FFS option	
T <sup>YYY</sup> Plan	time to plan for <b>YYY</b> *	
TYYY Trnsfr	time to move task data and code for YYY*	
$T_{CmpExec}^{YYY}$	time to complete task execution after <b>YYY</b> *	
T <sub>setup</sub>	time to establish interconnection network data path	
T <sub>xmit</sub>	time to transmit one word over interconnection network	
T <sub>message</sub>	time to send a message over interconnection network	
T <sub>DA</sub>	time to access disk	
$\eta_{exec}(x)$	estimated execution time for task on x PEs	
τ	total time spent executing task prior to associated checkpoint	
$F_{sm}(t)$	probability of submachine failure from time 0 to time t	
$R_{sm}(t)$	submachine reliability function, $R_{sm}(t) = 1 - F_{sm}(t)$	
$T_{Penalty}^{WC}$	worst case reconfiguration-option-choicepenalty	

\*YYY = FFS, TM, or TR reconfiguration option.

----



----

-

Figure 1: Fault-free subdivision recovery option.

fault has been located and the recovery point for the task has been determined, the SCU must make a determination about the partitionability of the current submachine. For this option, it is assumed that algorithmic constraints dictate that the subdivision is required to possess the same topological network properties as the original submachine. The recovery option of Section 6.3.3 is used when there is no need to preserve these properties. To create independent submachines with the same topological network properties as the original network, all submachines in a system possessing a multistage cube or hypercube interconnection network must have sizes that are a power of two [Sie80, Sie90]. Thus, if a submachine can be partitioned (i.e., it is not of minimum size), it can be partitioned into two equal-size submachines. For a machine with  $N = 2^n$  PEs, numbered from 0 to N – 1, the numbers of all PEs in a submachine of size  $K = 2^k$  agree in n-k bit positions. These n-k bits are called submachine bits. Without loss of generality, let these submachine bits be the low-order n-k bits. The fault-free subdivision (half) B of this submachine A is composed of K/2 PEs. B's submachine bits are calculated by appending the complement of bit n-k+l of the faulty PE number to the high-order end of submachine A's submachine bits. Thus, to determine the submachine bits of the fault-free subdivision (FFS) requires a constant amount of time T<sup>FFS</sup><sub>Check</sub>.

The number of physical processors in the subdivision is half the number of physical processors in the original submachine. Therefore, the virtual processor ratio for the subdivision will double. The SC must determine the new mapping of virtual processors onto physical processors and coordinate the relocation of program code and data accordingly.  $T_{Map}^{FFS}$ , the time to perform this mapping, is computable by the SC and will depend on the implementation specifics of virtual processors in the system.

The next step is to move all the task program code and data onto the fault-free subdivision as per the virtual processor to physical processor mapping determined above. This will require  $T_{Trnsfr}^{FFS}$  time. The amount of code a task consists of can be easily determined during task compilation. However, the amount of data associated with a task can change dynamically during task execution. For this reason, part of the information saved during every checkpoint operation is the amount of checkpoint data saved. Because every PE stores its checkpoint data into a different PE within the same submachine, no data is lost when a PE becomes faulty. It is assumed that the interconnection network is used to transfer data. This transfer can be accomplished without conflicting with inter-PE messages from other submachines because of the partitioning properties of the network. The time to accomplish the data transfer can therefore be determined and will depend on the amount of data and the virtual processor to physical processor mapping.

In the SPMD (Single Program - Multiple Data) restriction of MIMD mode [DaG85],

every PE in the submachine has a copy of the same program. Thus, in SPMD mode, program code does not have to be transferred and  $T_{Trnsfr}^{FFS}$  will consist only of the time required to transfer the data. However, for SIMD and **MIMD** modes, some program code will have to be transferred.

The structure of **SC/PE** and inter-SC connections (if they exist) varies among approaches to **multiple-SIMD** architectures (e.g., CM-2 [TuR88], MAP [Nut77a, Nut77b], PASM [SiS87], TRAC[LiM87]). Because of this, the time to do any needed transfer of SIMD programs when "moving" a SIMD task from one set of **PEs** to a subset or different set is highly machine dependent. Thus, in the discussions that follow, it is assumed as an upper bound time requirement that the SIMD program is reloaded from secondary storage whenever a reconfiguration occurs.

For general **MIMD** mode tasks, program code will have to be moved from **PEs** not in the subdivision to those that **are** in the subdivision. Furthermore, depending on the mapping of virtual to physical processors, program code may have to be moved among **PEs** in the subdivision. The interconnection network is used to perform this transfer of program code and the time to do this can be determined in the same way as for the transfer of data across the network. However, the MIMD programs that resided on the faulty PE will have to be loaded from secondary storage (disk).

The time to transfer a SIMD or MIMD program from disk depends on the disk latency, the amount of code to be transferred, and the bandwidth of the disk communication channel. In such cases,  $T_{Trnsfr}^{FFS}$  is difficult to predict accurately because of variation in disk latency from one access to another. Therefore, an expected time for disk access will have to be used to determine  $T_{Trnsfr}^{FFS}$ .

The **SCU** must then perform whatever operating system functions are needed to establish the new system partitioning. This requires a system dependent time represented by  $T_{Part}^{FFS}$ . The time to complete the task execution on the subdivision is  $T_{CmpExec}^{FFS}$  and will generally be greater than the time to complete the task execution on the original submachine. The difficulty of determining  $T_{CmpExec}^{FFS}$  for this recovery option and those that follow is discussed in Subsection 4.4.  $T_{Total}^{FFS}$ , the total time to reconfigure and complete a task on a subdivision of the original submachine is represented as follows.

$$T_{Total}^{FFS} = T_{Check}^{FFS} + T_{Map}^{FFS} + T_{Trnsfr}^{FFS} + T_{Part}^{FFS} + T_{CmpExec}^{FFS}$$

There are three main disadvantages of the subdivision recovery method. The first is that a subdivision of the current submachine may not exist, **i.e.**, there is no way to further subdivide the **current** submachine. This is due to restrictions on minimum submachine size, which is usually associated with SIMD operation. The second disadvantage is similar and follows from this. There is a limit to how many faults can be tolerated in this way because of the physical limits to the number of times a machine can be partitioned. Finally, two types of performance degradation may occur. The first is task performance degradation. A task will generally require more time to complete on a **subdivision** of the original submachine. This is further discussed in Subsection 4. The second type of performance degradation is that of the system. After subdividing a submachine because of a fault, the **entire** subdivision containing the faulty PE becomes idle. In **MIMD** mode, the fault-free **PEs** of this **subdivision** can be utilized by the system because the **PEs** act independently of one another. However, in SIMD mode, the system must repeatedly partition the subdivision containing the fault until the fault is in a submachine of minimum size. Other tasks can be executed on the fault-free submachines created during this partitioning process. If the minimum size of the submachine containing the fault is greater than one, fault-free **PEs** become underutilized and thus **contri**bute to a degradation in system performance.

# 3.2. Task Migration

It is likely that some tasks executing on a massively parallel processor will not use the entire machine and will therefore execute on an independent submachine formed by partitioning the machine. Reasons for partitioning include system utilization (e.g., due to the size of the data set to be processed, the task can make use of only a subset of the PEs), and reducing task execution time (some tasks can execute faster using fewer PEs [KrM88, SaS93, SiA92]). Therefore, if one or more PE faults occur in the current (source) submachine  $P_s$ , the task can be migrated to a another (destination) submachine  $P_d$ , if one is available. An example of this is provided in Figure 2. In this figure, PE 2 becomes faulty during the execution of a task on a submachine ( $P_s$ ) of four PEs. The task is then migrated to an idle submachine ( $P_d$ ) elsewhere in the system, and task execution is completed there. It is assumed that P, and  $P_d$  are of equal size. While this may not be a requirement for some systems, it is a reasonable assumption to make because P, was originally of an appropriate size for the task. A brief analysis of the task migration costs discussed in [ScS88] is provided below.

The first step to be made during the migration of a task from P, to  $P_d$  is to decide to which  $P_d$  to migrate. When more than one  $P_d$  is available, the  $P_d$  that results in the least cost of migration should be chosen. Two factors that enter into this decision are the locations of P, and  $P_d$  and the mapping of  $P_s$  PEs to  $P_d$  PEs. For a N = 2<sup>n</sup> PE machine, a O(n) time method for determining the mapping of PEs from P, to  $P_d$  that minimizes the task migration (TM) time is provided in [ScS90]. Thus,  $T_{Map}^{TM}$ , the time to choose  $P_d$  and to



Figure 2: Task migration recovery option.

determine the optimal mapping of source **PEs** to destination **PEs** is a function of the log of the size of the machine and the number of destination submachines to be considered. The **SCU** performs this function, because it is responsible for allocating and deallocating **sub**-machines.

The next step in the task migration process is to transfer all necessary task information from  $P_s$  to  $P_d$ . The time to accomplish the data transfer depends on a combination of the amount of data to be transmitted, the location of source and destination submachines, the source-PE-to-destination-PE mapping, the use of the interconnection network by other tasks, the type of network, and system implementation details. It is assumed that the interconnection network is used to transfer data for the SIMD case, and programs and data for all other cases. During the migration of a task, conflicts in the network may occur with other migrating tasks, or with normal inter-PE message **traffic** due to other tasks [**ScS90**]. These types of interference are not considered in the model presented here because: (1) it is assumed that two simultaneous migrations would rarely occur, and (2) interference with normal inter-PE message traffic is expected to be very limited relative to the task migration traffic.

Let  $T_{Trnsfr}^{TM}$  be the time to transfer the program and data. For multistage cube interconnection networks and with the exception of SIMD and general MIMD program code, the time to perform this transfer is a linear function of the amount of data to be transferred and the number of network permutation settings required to accomplished the transfer. The amount of time to transfer the SIMD program code to  $P_d$  is machine dependent. In the worst case, the code will have to be loaded from secondary storage into the SC of  $P_d$ . For the general MIMD case, the program for the faulty PE must also be transferred from secondary storage. As discussed earlier, this access to secondary storage makes prediction of the transfer time difficult.

As before,  $T_{CmpExec}^{TM}$  is the time to complete the task execution once the migration is complete.  $T_{CmpExec}^{TM}$  is equivalent to the time required to complete the task from the recovery point on the original submachine before the fault occurred. Thus,  $T_{Total}^{TM}$ , the time to migrate a task is as follows.

$$T_{Total}^{TM} = T_{Map}^{TM} + T_{Trnsfr}^{TM} + T_{CmpExec}^{TM}$$

The main advantages of task migration to another equal-size submachine **are** that no remapping of virtual processors to physical processors is required and once the migration is completed, the task will finish executing without any performance degradation. However, the overhead to move a task may be significant compared to the task execution time remaining. As with the fault-free subdivision option, system degradation in the form of underutilization of fault-free **PEs** may occur for the **PEs** in **P**,.

It is possible that no idle destination submachine exists to which to migrate a task. The task can be migrated to a submachine already being used to execute another task, and the two tasks can time-share the submachine. A number of factors should be considered in this event. First, it is desirable that the submachine to be shared have enough memory to hold both tasks. Second, if the expected completion times for tasks are known, the submachine with the earliest expected completion time should be selected. Finally, because two tasks are sharing one submachine, the estimated remaining execution time for each is more than doubled because of the context switching that must take place. Although time-sharing of **sub**-machines is not included in the quantitative framework presented here, the framework can be extended to include this option. To do this, it will be necessary to establish a method to incorporate into the reconfiguration choice the impact on the task already executing on the destination submachine.

# 3.3. Task Redistribution

In many cases when a PE or PE memory fault occurs, it may be possible to redistribute the data and/or subtasks associated with the faulty PE to neighboring PEs within the same submachine, thus effectively removing the faulty PE from the computation. Figure 3 illustrates this task redistribution recovery option. The ability to redistribute the task in such a way that the faulty PE is effectively removed from the computation depends on a number of factors including the algorithm, the mode of operation, and the interconnection network. Consider an algorithm that exhibits coarse grain parallelism and is implemented as a set of MIMD subtasks. If a PE becomes faulty, the subtask that was executing on that PE can be distributed to one or more fault-free PEs. However, to minimize the execution time of the task, load balancing may be required. Work on load balancing on parallel and distributed systems [e.g., NaM92] has shown that the optimal load balancing solution depends on a number of factors including the task/subtask queuing model used, the precedence constraints involved, and on the tasks being executed. Thus, it is unlikely, using available technology, that a distribution solution can be found that works for all classes of problems. Furthermore, there is a trade-off between the time it takes to distribute the load and the resulting savings in execution time.

Consider an SIMD machine with a mesh interconnection network. A faulty PE might require that all the data on all the **PEs** be considered in the redistribution to achieve optimal performance and preserve the near-neighbor communication pattern [UyR88]. If a

----



\_\_\_\_

Figure 3: Task redistribution recovery option.

multistage cube network is used, it is possible that the network will not be able to support the required single-pass inter-PE communication permutations on a submachine where one or more **PEs** have been effectively removed (i.e., the permutations permitted are limited **[Law75]**). In general, there may be additional overhead incurred whether extra time is required to redistribute the data or extra time is needed to perform communication in a network where a needed permutation cannot be performed in one step due to a PE fault.

The first step in the task redistribution (<u>TR</u>) process is to determine how to accomplish the redistribution. In general, an equal load distribution (of data for **SIMD/SPMD** tasks and of both programs and data for MIMD tasks) among all of the fault-free **PEs** will result in the greatest efficiency for completion of the task. The time to make the redismbution,  $T_{Map}^{TR}$ , depends on the mode of operation, the algorithm mapping, and the current state of each subtask.

It is assumed that the SC has user-supplied knowledge about the algorithm mapping, which it uses to decide how to redistribute the task. Ideally, this knowledge would be compiler-supplied. However, while this may be possible for certain classes of problems, it is still an open problem in the general case.

The next step is to perform the **subtask** and/or data redistribution.  $T_{Trnsfr}^{TR}$  is the time required to accomplish the relocation of the program code and/or data. As in the previous two options, the time to accomplish this transfer depends on the mode of operation, the amount of data to be transferred, the location of source and destination **PEs**, the interconnection network, and system implementation details. Once again, for a MIMD task, the transfer of program code from secondary storage will add some unpredictability into the expected transfer time. The time to complete task execution after task redistribution,  $T_{CmpExec}^{TR}$ , is expected to be greater than the time to complete the task on the fault-free submachine and is discussed further in Subsection 4.4. Thus,  $T_{Total}^{TR}$ , the time to redistribute a task among the fault-free **PEs** of a submachine and complete execution of the task is as follows.

$$T_{\text{Total}}^{TR} = T_{Map}^{TR} + T_{Trnsfr}^{TR} + T_{CmpExec}^{TR}$$

An advantage of the task redistribution option is that all the fault-free **PEs** in the **sub**machine can be utilized by the task, while with the fault-free subdivision option, only half the number of original **PEs** are utilized. A disadvantage is that to allow data and **subtasks** to be redistributed with a minimum of effort, the task must be coded and/or compiled considering fault tolerance and reconfiguration. As an example, linked lists should be favored over indexed arrays when random access to the list elements is infrequent, because the code does not have to know the number of elements in the structure and does not have to be modified when the data is redistributed. **Of** the three options for reconfiguration discussed thus far, the task redistribution option is by far the most difficult to quantify. However, if the task redistribution option is to be considered, the information discussed in this section is needed to choose the best alternative.

#### 3.4. Using Remote Memory for the Faulty PE

Indirect addressing using base address registers or index registers is a commonly employed addressing mode in processor designs (e.g. Motorola 68000 family, Intel 80X86 family). Using this addressing mode, a register contains the address of an operand. The register is incremented or used with an offset to access other operands. Thus, the same program can access different blocks of data in memory, depending on the contents of one register. Because the classes of distributed shared memory parallel processing systems under consideration have a single shared logical address space, it is possible for the processor of a PE that has experienced a memory fault to continue execution using remote memory locations exclusively.

As stated earlier, the checkpoint data for every PE in a submachine is stored in the memory of a different PE within the same submachine. Thus, no data is lost when a memory module fails. For all modes of execution, it is assumed that the PE with the faulty memory module will access the checkpoint data remotely.

Because in SIMD mode only data is stored in PE memory, execution can continue after a memory fault once the registers used to access operands in the faulty PE are loaded with the addresses of the checkpoint data for that PE. Also, because all the processors operate in lockstep, assuming that a remote memory access requires more time than a local memory access, there will be some increase in execution time when one or more processors are required to make a remote memory access as opposed to all processors performing a local memory access. This impact on execution time is quantified in Sections 6.4 and 6.5. Single-pass network permutations possible on a fault-free submachine are still possible when one or more processors use only remote memory. This is because it is assumed that all the PE processors utilize transmit and receive registers for network communication in SIMD mode. A memory failure will not affect how these registers are used by the network.

In MIMD mode, to maximize performance, program code and data are typically located in the **PEs** where they are needed. If the processor of a PE with a memory fault is required to fetch all its instructions and data from a remote memory as opposed to local memory, one would expect a significant degradation in the performance of that processor. Processors that possess a local cache **and/or** that perform instruction prefetching would be expected to suffer a smaller performance degradation in most cases. These factors are beyond the scope of this model, but the model can be extended to include them. Such extensions and the associated analysis would be necessary to make use of the model with this **type** of processor.

Another complication that arises in the MIMD mode of operation is that the lack of synchronization among processors can result in interconnection network conflicts. If one or more processors are required to fetch all instructions and data from remote memory, the likelihood of interconnection network conflicts is increased. A local processor cache would decrease the impact of this. Instruction prefetch capability would not decrease network **traffic**, but the degradation due to network conflicts may be reduced by the overlapping of instruction fetch and execution. Modeling the amount of overhead incurred as a result of network conflicts is very difficult because of its dependence on the system task load, the input data for each task, and the system architecture and hardware implementation. For simplicity, the model presented here assumes a negligible overhead due to communication network collisions. Further work is needed in this area to develop a more complete model.

Regardless of the execution mode, the data of the faulty PE is already stored in a remote memory during the checkpointing process. For SPMD mode tasks, every PE has an identical copy of the task code. In this case the processor of the faulty PE can access another PE's copy of the code remotely. Thus, the only case requiring a transfer of program code is the general MIMD mode case. In this case, the **subtask** code for the faulty PE must be loaded from secondary storage to remote memory (*RM*). The time to do this is  $T_{Trnsfr}^{RM}$  and is a function of a number of factors as discussed earlier, including the disk latency ( $T_{Trnsfr}^{RM} = 0$  for SIMD  $\alpha$  SPMD mode). As before, the disk latency is difficult to predict and requires the use of an expected value.

Once the code for the faulty PE has been relocated, the SC must write the proper base addresses to the faulty PE's base registers. This will require  $T_{Reg}^{RM}$  time, which will vary with the number of registers that must be modified. The time required to complete the task execution after reconfiguration is  $T_{CmpExec}^{RM}$  and will generally be greater than the time required on the fault-free submachine. Quantifying  $T_{CmpExec}^{RM}$  is discussed in detail in Sections 6.4 and 6.5. Thus,  $T_{Total}^{RM}$ , the time to prepare a task to continue execution substituting remote memory for the local memory of a faulty PE and to complete the task execution is as follows.

$$T_{Total}^{RM} = T_{Trnsfr}^{RM} + T_{Reg}^{RM} + T_{CmpExec}^{RM}$$

In addition to the time required, there is a memory space requirement to consider as

well. The use of base register addressing requires that there be a contiguous block of memory large enough to hold the data structure pointed to by a base register. Therefore, some remote memory module is required to have enough contiguous memory free to hold the faulty **PE's** program code (in addition to the checkpoint data it must hold for some PE). The advantage of the remote memory approach is that all the **PEs** from the original submachine are utilized and no changes to the task code are necessary. A disadvantage is that the processor forced to use only remote memory will generally execute instructions at a slower rate and may interfere with the use of the interconnection network by other **PEs**.

#### 4. Data-Independent-Execution-Time Model

One recovery cost common to all the options presented is the remaining execution time. However, the time required to complete execution of a task varies with the recovery option. For example, a task that has nearly completed executing may complete much earlier on a subdivision than it would if it were migrated to another submachine, especially if a high overhead migration cost is incurred. Alternatively, if a significant amount of computation remains, migrating the task may result in an earlier task completion time. Thus, the remaining task execution time becomes important.

Tasks can be divided into two categories based on execution time. These are tasks with data-independent execution times and tasks with data-dependent execution times. A task with a <u>data-independent execution time</u> does not depend on input data to make branching decisions. Thus, the number of times any branch in the task program code is taken can be determined by a compiler during program compilation, and a compiler can determine an expected execution time for the task. In contrast, a task with a <u>data-dependent execution time</u> has branch decisions that are based on data that is known only at run time. The execution time of such tasks is considered in the next section.

A model is presented in this section for determining the execution-time costs of dataindependent-execution-time tasks. This model differs from other models in that it can accommodate the remote memory recovery option. Such an execution-time model must necessarily include detail about the amount of time spent making local and remote memory accesses. Both SIMD and MIMD versions of the model are presented. Because SPMD is a subset of MIMD, it is accommodated by the **MIMD** model. Mixed-mode operation execution-time prediction can be accomplished by extending the model to account for switching between modes.

#### 4.1. SIMD Task Execution Time

In SIMD mode, it is assumed that instructions to be executed by the submachine PEs are loaded into an automatic broadcast queue by the SC. This leaves the SC free to perform computations of its own (e.g., the manipulation of loop index variables, PE-common array index calculations). Thus, the execution time for an SIMD task includes the SC execution time plus the PE execution time minus any overlap between the two. For a comprehensive model for determining the amount of SC/PE overlap in an SIMD task see [KiN91]. Let  $T_{SC}$  be the time the SC spends performing its calculations,  $T_{OL}$  be the total SC/PE overlap, and  $T_{PE}(P,i) = 0$ . In SIMD mode, a new instruction i on processor P. For a disabled PE,  $T_{PE}(P,i) = 0$ . In SIMD mode, a new instruction is broadcast to the PEs by the SC only after the current instruction's execution has been completed (or initiated, if prefetching is used). Therefore,  $T_{SIMD}$ , the overall time for a SIMD task consisting of the execution of I instructions on Q PEs,  $0 \le P \le Q - 1$ , is

$$T_{SIMD} = T_{SC} - T_{OL} + \sum_{i=1}^{l} \max_{P} \left[ T_{PE}(P,i) \right]$$

An extension to this SIMD execution-time model is desired to adapt it for use in analyzing the difference in execution time when all local memory references are replaced by remote memory references. Only PE memory faults are of interest here, so the SC execution-time component is not affected. However, the SC/PE overlap component may be affected because it is a function of PE execution time. Let  $t_{LOC}$  be the time required for a PE to make a single access to its local memory. It is assumed that any such access takes a constant amount of time. Similarly, let  $t_{REM}$  be the average time required to make an access to a remote (non-local) memory location.

For the architectures under consideration, a remote memory reference requires use of the interconnection network. For a read of remote memory, an address must traverse the network to the remote memory and the memory contents must traverse the network to the source of the read request. For a remote memory write, an address and word to be written must be sent to the remote memory location. Generally, an acknowledgment of some type is returned by the remote network interface. The time required for an address or data word to traverse the network can be divided into two components. These are transfer time and overhead. The transfer time is the actual propagation time required assuming no contention in the network. Transfer time is a constant for networks that have equidistant paths between any source and destination pair (e.g., multistage cube network), but may vary for other networks (e.g., hypercube). The overhead is time spent to establish a path through the network and the time spent waiting due to network contention or contention at the remote memory. In general, the determination of overhead is a difficult problem that depends on the network implementation details and the message traffic. Thus,  $t_R$  is only **an** estimated average value; calculating an exact time for each transfer would require machine and application-&pendent analysis. Research on &tailed modeling of networks has been the topic of entire papers, (e.g., [Har91]), and is beyond the scope of this work.

Here, for the sake of simplicity, remote memory reads and writes are assumed to take the same amount of time. The model could be extended to include different costs for the two types of references.

The number of local and remote memory references made by PE P during instruction i (when no PE is faulty) is denoted by  $M_{LOC}(P,i)$  and  $M_{REM}(P,i)$ , respectively. A compiler could determine these values for every instruction in a task as well as the amount of time spent by PE P on instruction i doing non-memory reference work ("computation time"),  $T_{COMP}(P,i)$ . For a disabled PE,  $M_{LOC}(P,i) = 0$ ,  $M_{REM}(P,i) = 0$ , and  $T_{COMP}(P,i) = 0$ . Thus, the total time spent by PE P on instruction i is given by

# $t_{LOC}M_{LOC}(P,i) + t_{REM}M_{REM}(P,i) + T_{COMP}(P,i).$

Therefore, the new SIMD execution-time model becomes

$$T_{SIMD} = T_{SC} - T_{OL} + \sum_{i=1}^{I} \max_{P} \left[ t_{LOC} M_{LOC}(P,i) + t_{REM} M_{REM}(P,i) + T_{COMP}(P,i) \right]$$

To determine the execution time for a task on a submachine in which one or more **PEs** have faulty local memories, the following modifications are made to the faulty PE memory reference counts.

$$M_{REM}(P,i) \leftarrow M_{REM}(P,i) + M_{LOC}(P,i); M_{LOC}(P,i) \leftarrow 0$$

The model for determining SIMD task execution time presented above can be applied in cases where the exact sequence of instructions to be executed is known at compile time. However, for most programs the sequence of instructions executed is data dependent. For such programs it is not possible to determine an execution time during compilation.

Furthermore, it is not practical for the SC to determine the amount of execution time left when a fault occurs, because it would potentially have to simulate the execution of every instruction on every PE to determine when the task has completed. In cases where empirical data is available on the execution time of a particular task, it is possible to determine an estimate of the execution time remaining. Such an estimate is discussed in the next section. For data-dependent-execution-time tasks where there is little empirical data, there is no practical way of determining an execution time. In such cases, the reconfiguration decision may have to be made without benefit of execution-time knowledge.

# 4.2. MIMD Execution Time

It is assumed in the following model that an instruction cycle is composed of a fetch phase and an execute phase. Because each PE may execute a different program in MIMD mode, let I(P) denote the number of instructions that PE P must execute to complete a task and let  $T'_{PE}(P,i)$  be the time for PE P to execute its  $i^{th}$  instruction. Then,  $T_{MIMD}$ , the execution time for a task in MIMD mode is

$$T_{MIMD} = \max_{P} \left[ \sum_{i=1}^{I(P)} T'_{PE}(P,i) \right].$$

This ''max of sums'' for MIMD versus ''sum of max's'' for SIMD has been discussed in another context in [BeK91].

Let W(P,i) represent the number of words that PE P must fetch from local memory to read in instruction i. As before, the number of local and remote memory references made by PE P during its  $i^{th}$  instruction is denoted by  $M_{LOC}(P,i)$  and  $M_{REM}(P,i)$  respectively. A Compiler could determine these values for every instruction in a task as well as  $T_{COMP}(P,i)$ , the amount of time spent by PE P on its  $i^{th}$  instruction doing non-memory-reference work. Thus, the MIMD task execution time becomes

$$T_{MIMD} = \max_{P} \left[ \sum_{i=1}^{I(P)} \left[ t_{LOC} \left[ M_{LOC}(P,i) + W(P,i) \right] + t_{REM} M_{REM}(P,i) + T_{COMP}(P,i) \right] \right].$$

To determine the execution time for a task on a submachine in which one or more **PEs** have faulty local memories, the following modifications are made to the faulty PE memory

reference counts.

$$\begin{split} M_{REM}(P,i) \leftarrow M_{REM}(P,i) + M_{LOC}(P,i) + W(P,i); \\ M_{LOC}(P,i) \leftarrow 0; \ \mathrm{W}(\mathrm{P},\mathrm{i}) \leftarrow 0 \end{split}$$

As with the SIMD execution-time model, the equation above is useful only when the exact sequence of instructions for every MIMD **subtask** is known. A further consideration in using remote memory in place of faulty local memory is that the faulty PE must fetch all its instructions from remote memory, which will cause a significant increase in the amount of interconnection network traffic. Therefore, the probability of conflicts in the network is also increased. The MIMD execution-time equation given does not account for conflicts in the network, so the execution time derived from the equation will be a minimum execution time.

#### 5. Data-Dependent-Execution-TimeModel

Most tasks have a data-dependent execution time, making the execution-time model presented in the preceding section inapplicable. However, in a production environment where a task is executed repeatedly on various sets of data (e.g., image processing of satellite pictures), empirical studies can be performed to derive an estimated execution time for a task. For these cases it is possible to develop a model that will predict the expected execution time remaining for a task that is recovering from a fault.

#### 5.1. Execution Time When Using a Different Submachine

Once a task has been migrated from a submachine containing a faulty PE to a fault-free submachine of equal size, the time to complete the task execution will be the same as it would have been on the original fault-free submachine. An estimate of this expected completion time is now derived.

Let  $\hat{T}_{Exec}$  be a discrete random variable that represents the execution time for a specific task.  $T_{Exec}$  is always positive because a negative execution time is not possible.  $f(T_{Exec})$  is the discrete density function [Pap84] of  $\hat{T}_{Exec}$ . If  $\hat{T}_{Exec}$  takes the value  $x_i$  with probability  $p_i$ , and  $\delta(x)$  is the impulse function such that  $\delta(x - x_i)$  has value one at x = xi and value zero elsewhere, then

$$f(T_{Exec}) = \sum_{i} p_i \delta(T_{Exec} - x_i).$$

By definition, the expected value of  $\hat{T}_{Exec}$  is

$$E\left\{\hat{T}_{Exec}\right\} = \int_{0}^{\infty} T_{Exec} f(T_{Exec}) dT_{Exec} = \eta_{exec}.$$

It is assumed that the amount of execution time spent on a task prior to a checkpoint is stored with that checkpoint. If a recovering task is to proceed from a checkpoint and the execution time stored with that checkpoint is  $\tau$ ,  $T_{CmpExec}$ , the expected amount of time required to complete task execution is

$$T_{CmpExec} = \eta_{exec} - \tau.$$

 $T_{CmpExec}$  assumes that the submachine size remains the same and that all the PEs in the submachine are fault-free.

#### 5.2. Execution Time on a Fault-Free Subdivision

Here, the completion time of a task that completes execution on a subdivision that is half the size of the original submachine is considered. Let  $T_{2x}$  be the **expected** execution time of a task on a submachine consisting of 2x PEs, and let  $T_x$  be the expected execution time of the same task remapped onto one of the two equal-sized subdivisions of the original submachine. If the average time for an inter-PE transfer remains the same in either case,

$$T_{2x} \geq \frac{1}{2}T_x.$$

The inequality becomes an equality when the mapping of the task from the subdivision onto the 2x PE submachine is optimal.

This equation can be extended to submachines of arbitrary size as follows. Let  $S_{Old}$  be the number of **PEs** in the old (original) submachine and let  $S_{New}$  be the number of **PEs** in the new submachine (subdivision). Then, the expected remaining execution time on the

subdivision is bounded by

$$T_{CmpExec} \leq \frac{S_{Old}}{S_{New}} \left[ \eta_{exec} - \tau \right].$$

A more accurate remaining execution time estimate can be obtained if empirical data is available to determine expected task execution times for the submachine sizes of interest. Then, the estimated task execution time becomes a function of the submachine size and the estimate of the remaining execution time becomes

$$T_{CmpExec} = \eta_{exec}(S_{New}) \left[ 1 - \frac{\tau}{\eta^{exec}(S_{Old})} \right].$$

# 5.3. Execution Time After Task Redistribution

An execution-time estimate for the task redistribution recovery option is more difficult than for the previous options. Consider a task executing on a submachine of size  $S_{Old}$  in MIMD mode. If a PE becomes faulty and its **subtasks** are distributed equally to the  $S_{New} = S_{Old} - 1$  fault-free **PEs** in the submachine, the remaining execution time is bounded as follows.

$$T_{CmpExec} \leq \frac{\text{Sold}}{\text{Sold} - 1} \left[ \eta_{exec} - \tau \right]$$

Consider the case where the faulty **PE's subtasks** cannot be distributed equally among the fault-free **PEs**. In the worst case, all the faulty **PE's subtasks** would be assigned to one PE. In this case, the remaining execution time could be twice that of the remaining execution time on a fault-free submachine. The degree to which the performance of the system on a task is degraded is a function of the system (a), the algorithm implementation ( $\beta$ ), and the number and location of faulty **PEs** (y). Let  $d(\alpha, \beta, \gamma)$  be defined as the amount of performance degradation;  $1 < d(\alpha, \beta, \gamma) \le 2$ . The amount of performance degradation can be estimated from empirical data or from user provided information about the algorithm implementation. The remaining execution time then becomes

$$T_{CmpExec} = d(\alpha, \beta, \gamma) \left[ \eta_{exec} - \tau \right].$$

#### 5.4. Execution Time When Using Remote Memory

Estimating the remaining execution time for a task executing on a submachine that includes one or more **PEs** using remote memory exclusively is now explored. Because the tasks under consideration have data-dependent execution times, it is not possible, before run time, to determine exactly how many times any data-dependent branches (including those necessary for loops and conditionals) in a program are executed. However, in the production environment assumed, code profiling can be used to determine expected values for the number of times each branch is taken. If such empirical data is not available, the user must estimate these expected values if the model presented here is to be used.

Given expected values for the number of times each branch is taken in a datadependent-execution-time task, one can calculate the remaining execution time for the task after fault recovery as if it were a data-independent-execution-time task. The last valid checkpoint before the occurrence of the fault is used to determine the state from which the task execution will begin. As before, where accesses to local memory were used in the original **program(s)**, remote memory accesses will be made instead. For SPMD and MIMD modes, the fetching of instructions from local memory will be replaced with fetches from remote memory. However, for SIMD mode, instruction fetching is not included because instructions are broadcast to the **PEs** by the SC.

Using the execution-time models presented in this section, one can arrive at an estimate of the remaining execution time for a task after recovery from a fault. The remaining execution time for tasks with data-dependent execution times for which no empirical data is available cannot be estimated reliably. In these cases, the user must be required to supply an estimate prior to execution time if a comparison of recovery options is to be performed.

An alternative method for determining remaining execution time centers around the use of an automatic complexity evaluator such as that presented in **[LeM88]**. The approach here is to attempt to generate a **nonrecursive** function, prior to run time, that can be solved at run time to determine the asymptotic time-complexity behavior of a program. This approach is applicable to a wide range of programs, but it is not always successful in generating a **nonre**cursive function. Further study is needed to see how it can be applied here.

# 6. Choosing an Option

#### 6.1. Overview

Thus far, general quantitative models of four different reconfiguration schemes have been presented. It was pointed out that collecting precise values for some of the model parameters is very difficult (if not impossible). The next step is to analyze the information available to determine if guidelines can be developed for making a choice among these methods for practical implementations. For the remainder of the chapter, only the fault-free subdivision, task migration, and task reconfiguration recovery options are considered. Work is ongoing to incorporate the remote memory recovery option into the analyses that follow.

The time to reconfigure and complete a task for the fault-free subdivision, task migration, and task redistribution reconfiguration options can be separated into three primary components: time to plan for the reconfiguration option ( $T_{Plan}$ ), time to move the task data and code ( $T_{Trnsfr}$ ), and time to complete the task ( $T_{CmpExec}$ ).



In this section, the relative impact these three components has on the overall reconfiguration cost is discussed. Experimentally determined ranges for these parameters on the **PASM** prototype and the **nCUBE** 2 are used in the analysis where applicable.

# 6.2. Range of T<sub>Plan</sub>

First, consider  $T_{Plan}$  on PASM for the fault-free subdivision option. The time required for the PASM SCU to determine a fault-free subdivision,  $T_{Check}^{FFS}$ , is a constant for a fixed-

sized machine (in terms of number of **PEs**). In PASM with 2" **PEs**, the numbers of all the **PEs** in a submachine of  $2^k$  **PEs** agree in their low-order n-k bits; i.e., a partition of size 2' is uniquely specified by the common low-order n-k bits of the **PEs** in the partition. The logical PE numbers are the k most significant bits of the physical PE number. The fault-free subdivision is determined by the low-order n-k+1 bits of the faulty PE number with the (n-k+1)st bit complemented. This was determined, by experimentation, to require 32 microseconds (for the 16 PE prototype) on the PASM SCU.

For an **nCUBE** 2 with 2" **PEs**, a submachine is specified by an anchor node and submachine size  $(2^k)$ . The other **PEs** in the submachine can be found by XOR-ing the logical node numbers, which range from 0 to  $2^k$ -1, with the anchor PE number. Thus, the physical PE numbers in a **subcube** of size  $2^k$  always agree in their upper n-k bits and vary in their low order k bits. To find the anchor PE number for the fault-free subdivision, the (k-1)st bit of the original anchor PE number is replaced by the complement of the (k-1)st bit of the faulty PE number. The time to do this on a 64-PE **nCUBE** 2 was experimentally found to be 29 microseconds.

Having determined the fault-free subdivision  $(T_{Check}^{FFS})$  the next step is to map virtual **PEs** to the physical **PEs** in the fault-free subdivision. Neither the PASM machine or the nCUBE 2 actually implements virtual processors. However, the time to determine the mapping,  $T_{Map}^{FFS}$ , can still be determined based on the architectural characteristics of the machines. For PASM, a virtual PE that was on logical PE L is mapped to PE  $\lfloor L/2 \rfloor$ . Because the physical PE numbers of the **PEs** in the fault-free subdivision all have the same low-order n - k+1 bits in common (for a subdivision of size  $2^{k-1}$ ), and because the logical PE numbers are the high-order k-1 bits of the physical PE numbers, the mapping is known a priori. For nCUBE 2, a virtual PE that was on logical PE L is mapped to PE L mod  $2^{k-1}$ . As discussed earlier, the logical PE numbers for the nCUBE 2 are converted to physical PE numbers by XOR-ing them with the physical PE number of the anchor node (selected in  $T_{Check}^{FFS}$ ). Thus, the mapping is once again known a priori. Therefore  $T_{Map}^{FFS} = 0$  in both cases.

In the PASM system, only the SCU is aware of the machine partitions. Changing the system partitioning is performed by writing one word of data to a system partition register. The PASM **SCU** can use current overall machine partition information along with the fault location (to identify the submachine to be subdivided) to determine the correct data to write to the system partition register. For the PASM prototype, the time to generate the correct data word and write it to the partition register,  $T_{Part}^{FFS}$ , ranges from 36 to 48 microseconds, depending on the original system partition register contents.

For the nCUBE 2, two tasks must be completed to partition a submachine into two equal-size submachines. First, the **PEs** in the fault-free subdivision must be informed of the

-----

change in submachine size. This is required so they can properly determine their logical PE numbers. The time to do this is equal to the time it takes the **SCU** (front-end processor) to broadcast a short message to this effect to the **PEs**. The **nCUBE** 2 **PEs** are capable of implementing tree-structured communication paths for the broadcasting of data from the SCU. When there **are** no conflicts, establishing such a path requires only slightly more time (a few microseconds) than establishing a point-to-point path. Because the **PEs** in the fault-free subdivision **are** idle at this point, there can be no conflict when attempting to establish a broadcast path among the subdivision **PEs**. The second task is for the SCU to update its system tables to reflect the new machine partitioning. This can be performed in parallel with task execution on the fault-free submachine and is therefore not considered here. Because the **nCUBE** 2 SCU is actually a time-shared **UNIX** system, it is difficult to accurately determine  $T_{Part}^{FFS}$  in this case. However, it is expected that  $T_{Part}^{FFS}$  is on the order of the transfer of a short message over the PE interconnection network; in the range of 160 to 300 microseconds for the **nCUBE** 2 (discussed further in the next subsection).

Consider  $T_{Map}^{TM}$ . Here, a destination submachine must be selected for task migration. The choice should be one that minimizes the time to transfer the program code and data. Recall that on the PASM prototype, all the **PEs** in a submachine of size  $2^{k}$  agree in their n-k least significant bits, and the k most significant bits can then identify the logical PE number of a PE within the submachine. A constant offset permutation is a one-to-one and onto mapping of source PEs to destination PEs such that the physical PE number of any source PE minus the physical PE number of its associated destination PE yields the same constant (mod 2" for a machine with 2" PEs). If a logical PE in the source submachine is mapped to the same logical PE number in the destination submachine, the resulting inter-PE data transfer is a constant offset permutation, which is guaranteed to be conflict free for a multistage cube [Law75, Pea77]. Therefore, such a mapping is optimal and known in advance. Thus,  $T_{Map}^{TM}$ consists only of selecting any destination submachine of the desired size. It is assumed that the SCU maintains a table of available submachines. While the current prototype system does not provide for this, the time to do a table look-up to find a free submachine of  $2^k$  PEs can be roughly estimated to be in the range of 10 to 100 microseconds. Depending on the table implementation, more time could be required for larger systems, due to a greater number of table entries.

On the **nCUBE** 2, assume that the anchor node of the destination submachine is selected such that its low-order k bits are identical with the low-order k bits of the anchor node of the source submachine. Then, the physical numbers of **PEs** with the same logical numbers in the source and destination submachines will agree in their low order k bits and will differ in exactly the same high-order n-k bits. Thus, a conflict-free mapping exists between the **PEs** 

with the same logical numbers in the source and destination submachines. The number of links separating these submachines is equal to the Hamming distance between their corresponding n-k high-order bits, and can be no greater than n-k. It is assumed that the the number of links traversed has a negligible impact on the overall transfer time (see Subsection 4.3). As was the case for PASM, it is assumed that the **nCUBE** 2 SCU maintains a table of available submachines. Once again, the time to perform a table look-up is estimated to be in the range of 10 to 100 microseconds.

Now consider  $T_{Map}^{TR}$ . As stated earlier,  $T_{Map}^{TR}$  depends on the mode of operation, the algorithm mapping, and the current state of each **subtask**. It is assumed that the SC has **user-suplied** (or possibly compiler-supplied) knowledge about the algorithm mapping, which it uses together with the faulty PE number to decide how to redistribute the task. It is further assumed that for most tasks, the decision can be programmed (by the user or by an automated process) in the form of a "C" switch statement. The faulty PE number is merely a variable in the program that directs the redistribution of program code and data. Thus, the decision can be made in the time that the switch statement can be evaluated (estimated to be 1 to 10 microseconds for PASM and **nCUBE** 2). It is possible that the mapping of data to **PEs** will change several times during the execution of a program. In such programs, the decision of how to redistribute the data may also depend on the mapping that was in place when the fault occurred. Although  $T_{Plan}^{TR}$  will require more time in such cases, it is expected that  $T_{Plan}^{TR}$  will still be less than  $T_{Trnsfr}^{TR}$  (discussed in the next subsection).

# 6.3. Range of T<sub>Trnsfr</sub>

As discussed in Section 6.3,  $T_{Trnsfr}$  represents the amount of time required to move the task program code and data as prescribed by the reconfiguration option chosen. Here, the range for  $T_{Trnsfr}$  is considered for each reconfiguration option. Data collected on PASM (possessing a multistage cube interconnection network) and on nCUBE 2 (possessing a hypercube interconnection network) are incorporated into the analyses.

It is assumed that one data item calculated and stored during the creation of each checkpoint is the word count for that checkpoint. Given this word count, a mapping of source **PEs** to destination **PEs**, and knowledge of the interconnection network used in the system, it is possible to determine the expected time to transfer a **PE's** checkpoint data. The networks considered in the following discussion **are** a circuit-switched multistage cube network and a circuit-switched hypercube network. However, the analyses can be extended to other interconnection network implementations.

For circuit-switched networks, the time to transmit a message can be decomposed into

two phases: set up (path establishment) and actual data transfer. During the set-up phase, the message request (routing tag) must propagate through interchange boxes (multistage cube) or any possible intermediate nodes (hypercube) to establish the desired data path. This requires  $T_{setup}$  time. During the data-transfer phase, the actual data is transferred one word at a time and each word requires  $T_{xonit}$  time to traverse the path. Thus, the time to transmit a w-word message,  $T_{message}$ , is:

$$T_{message} = T_{setup} + w \times T_{xmit}$$

The above equation assumes that there are no delays due to network conflicts during the setup phase. The effect of network conflicts is discussed later in this section.

Figure 4 shows a plot of number of words transferred versus time required for the PASM experimental prototype in both SIMD and MIMD modes. The times shown include a small amount of loop overhead for each word transferred in MIMD mode.. The same overhead is present in the SIMD mode implementation, but it is executed by the SC and is overlapped with the data transfers being performed by the **PEs**. From these experiments, it can be seen that the time, in microseconds, to transmit a message on PASM is given by  $240 + w \times 24.6$  in SIMD mode and  $240 + w \times 123.4$  in MIMD mode. The reason for the large difference in times between the modes is due to the current implementation of message passing in MIMD mode on PASM. At present, only one word is sent at a time and the sending and receiving **PEs** must synchronize once for every word transferred. In **SIMD** mode, no synchronization is necessary because the **PEs** operate in lock-step. An MIMD implementation of message passing that requires one synchronization per message, rather than per word, would be much closer to the SIMD **performance**.

Figure 5 shows a plot of number of words transferred versus time required for the **nCUBE** 2. On the **nCUBE** 2, the transmission time depends, to a small degree, on the number of links (hops) that a message must traverse to travel from the sending PE to the receiving PE. Furthermore, the **nCUBE** 2 uses message buffers at the destination **PEs**. The maximum length of a message is determined by the size of the allocated **message** buffer. It is assumed here that the message buffer is 1024 words in length because this represents a good trade-off of local memory usage versus message-passing performance. Longer messages must be segmented into 1024 word blocks. The time to transmit a message composed of 1024-word blocks, where the last block may consist of less than 1024 words, is:



Figure 4: Message transmission time  $(T_{message})$  on PASM in SIMD and MIMD mode.

$$\left\lfloor \frac{w}{1024} \right\rfloor (T_{setup} + 1024 T_{xmit}) + T_{setup} + (w \ mod \ 1024) T_{xmit}$$

On the nCUBE 2, experiments yielded approximate values for  $T_{setup}$  and  $T_{xmit}$  of 160 microseconds and 0.57 microseconds, respectively.

Consider  $T_{Trnsfr}^{FFS}$ , the time to move a task's program code and data onto a fault-free subdivision. For both PASM and nCUBE 2, a careful choice of buddies can eliminate any need to transfer checkpoint data for the fault-free subdivision option. Specifically, the mapping of logical PE numbers discussed in the context of determining  $T_{Map}^{FFS}$  (in Subsection 4.2) can also be used to select buddies for checkpointing. In this case, the PEs in the fault-free subdivision will already have the checkpoint data from the PEs in the subdivision containing the faulty PE, and no transfer of checkpoint data is required.

For SPMD tasks, no program code has to be transferred because the **PEs** in the fault-free subdivision already have copies of the program. For MIMD tasks, the interconnection network is used to transfer the program code for the fault-free **PEs** in the subdivision containing the faulty PE. In this case, there can be no network conflict with tasks executing on other submachines because of the partitioning properties of the network. Also, as discussed in the previous subsection, the mapping of **PEs** used guarantees that only one network setting is required to **perform** the transfer. For SIMD tasks, the program code for the faulty PE for MIMD tasks. Thus, in these cases,  $T_{Trnsfr}^{FFS}$  includes the time to access disk,  $\underline{T_{DA}}$ .  $T_{DA}$  depends on a number of factors including the disk latency, the amount of code to be transferred, and the bandwidth of the communication channel. Physical limitations dictate that  $T_{DA}$  will require at least **10** milliseconds on average. The upper bound for  $T_{DA}$  is dependent on the size of the program to be moved (no larger than the size of PE memories).

Now, consider  $T_{Trnsfr}^{TM}$ . Because of the restrictions placed on the formation of submachines, i.e., all the physical numbers of the **PEs** in a submachine agree in their low-order or high-order n-k bits (see Subsection 4.2), only one network setting is required for the transfer between the **PEs** of the source and destination submachines for both PASM and nCUBE 2. However, in both cases, an additional network setting is required to transfer the checkpoint data of the faulty PE to the destination submachine because the faulty **PE's** checkpoint data is located in its buddy PE in the source submachine. Thus,  $T_{Trnsfr}^{TM}$  must be greater than  $2T_{setup}$ . In addition, the program code for SPMD tasks and for the fault-free **PEs** of **MIMD** tasks must also be transferred across the interconnection network. The program code for the destination submachine SC and the faulty PE will have to be loaded from



Figure 5: Message transmission time  $(T_{message})$  on nCUBE 2 for distances of one and four hops.

secondary storage for SIMD and MIMD tasks, respectively.

Unlike the fault-free **subdivision** option, during the task migration option there may be additional time required for the interconnection network transfers due to network conflicts with tasks on other submachines in the system. For this reason, and because program code is also transferred for SPMD tasks,  $T_{Trnsfr}^{TM}$  will always be greater than or equal to  $T_{Trnsfr}^{FFS}$ , i.e.,

$$T_{Trnsfr}^{FFS} \leq T_{Trnsfr}^{TM}$$

Finally, consider  $T_{Trnsfr}^{TR}$ . As with the fault-free subdivision option, the partitioning properties of multistage cube and hypercube interconnection networks guarantee that there is no network conflicts with tasks on other submachines during the redistribution of program code and data for the redistribution option. In addition, no program code has to be moved for SIMD or SPMD tasks. In general, SIMD and SPMD tasks have very regular distributions of data. In these cases, the redistribution of data will generally involve only a subset of the checkpoint data. For MIMD tasks, only the MIMD procedures associated with the faulty PE have to be transferred. Therefore, in the best case,  $T_{Trnsfr}^{TR}$  will be less than  $T_{Trnsfr}^{FFS}$ . In every case, at least one network setting is required ( $T_{Trnsfr}^{TR} > T_{setup}$ ). In some cases, the redistribution of program code and data among the fault-free PEs in a submachine may require a number of network settings. Thus, in the worst case,  $T_{Trnsfr}^{TR}$  may be greater than  $T_{Trnsfr}^{TM}$ , depending on the number of network settings required for task redistribution and on the number of conflicts encountered during task migration.

The ranges for  $T_{Plan}$  and  $T_{Trnsfr}$  on PASM and nCUBE 2 are summarized in Table II. It can be seen from the table that  $T_{Plan}$  is expected to require less than 400 microseconds for the systems considered. In contrast,  $T_{Trnsfr}$  can range anywhere from 0 to hundreds or thousands of milliseconds, depending on the amount of data being transferred across the interconnection network and from disk. The number of words transferred in both cases is bounded by the size of the PE memories. These ranges are compared to the expected range of values for  $T_{CmpExec}$  in Section 6.7. A coarse, but useful, range of values for  $T_{CmpExec}$  for each of the considered reconfiguration options is examined in the next subsection. These ranges for all three parameters are then compared and combined to determine an ordering of reconfiguration options based on the time needed to complete execution of the task after reconfiguration.

Approximate ranges, in microseconds, for  $T_{Plan}$  and  $T_{Trnsfr}$  for the FFS, TM, and TR reconfiguration options. Table II:

Option	<i>T<sub>Plan</sub></i> (PASM, nCUBE 2)	<i>T<sub>Trnsfr</sub></i> • (PASM, nCUBE 2)
FFS	$(68 \rightarrow 80, 189 \rightarrow 329)$	(0**, 0***)
ТМ	$(10 \rightarrow 100, 10 \rightarrow 100)$	(> 480 <sup>***</sup> , > 320 <sup>***</sup> )
TR	$(1 \rightarrow 10, 1 \rightarrow 10)$	(> <b>240</b> <sup>**</sup> , > 160 <sup>***</sup> )

\* Upper bound determined by size of PE memories.
\*\* Add *T<sub>DA</sub>* for SIMD, MIMD, or mixed-mode tasks.
\*\*\* Add *T<sub>DA</sub>* for MIMD tasks.

-----

# 6.4. Range of T<sub>CmpExec</sub>

A task with a data-independent execution time does not depend on input data to make branching decisions. Thus, the number of times any branch in the task program code is taken can be determined by a compiler during program compilation, and a compiler can determine an expected execution time for the task. In contrast, a task with a data-dependent execution time has branch decisions that are based on data that is known only at run time. In this case, it is assumed that an estimated execution time for the task can be determined through the use of empirical studies (i.e., information about task execution time on various sets of data), an automatic complexity evaluator such as that presented in [LeM88], or through analysis of the algorithm and data sets.

For all the reconfiguration options discussed, the number of **PEs** assigned to a task after the reconfiguration is equal to or less than the number of **PEs** originally assigned to the task. It is assumed that the average time for an inter-PE transfer does not increase when the task is executed on fewer **PEs**.

Once a task has been migrated from a submachine containing a faulty PE to a fault-free submachine of equal size, the time to complete the task execution is the same as it would have been on the original fault-free submachine. Let  $\underline{\eta_{exec}(2^k)}$  be the estimated execution time for a task on a submachine with  $2^k$  PEs. It is assumed that the total amount of execution time spent on a task prior to a checkpoint is stored with that checkpoint. If a recovering task is to proceed from a checkpoint and the execution time stored with that checkpoint is  $\underline{\tau}$ , the expected amount of time required to complete task execution after migrating the task to another submachine is

$$T_{CmpExec}^{TM} = \eta_{exec}(2^k) - \tau$$

 $T_{CmpExec}^{TM}$  assumes that the submachine size remains the same and that all the **PEs** in the submachine are fault-free. The expected range for  $T_{CmpExec}^{TM}$  is

$$0 < T_{CmpExec}^{TM} \leq \eta_{exec}(2^k).$$

Now, consider the completion time of a task that completes execution on a subdivision that is half the size of the original submachine.  $T_{CmpExec}^{FFS}$  is bounded as follows:

$$T_{CmpExec}^{FFS} \leq 2(\eta_{exec}(2^k) - \tau) = 2T_{CmpExec}^{TM}.$$

In addition, it is expected that  $T_{CmpExec}^{FFS} > T_{CmpExec}^{TM}$  because the number of processors in a fault-free subdivision is assumed to be half the number that would be available if the task was migrated to another submachine. Although some tasks can execute faster on fewer PEs [KrM88, SaS93, SiA92], it is assumed here that the original submachine size was selected for minimum execution time. That is, if a smaller submachine could be used to execute the task in the same or less time, the task would have been mapped to that smaller size submachine initially.

A more accurate remaining execution time estimate can be obtained if  $\eta_{exec}(2^{k-1})$  is known. Then, the estimated task execution time becomes a function of the submachine size and the estimate of the remaining execution time becomes

$$T_{CmpExec}^{EFS} = \frac{\eta_{exec}(2^{k-1})}{\eta_{exec}(2^k)} \left[ \eta_{exec}(2^k) - \tau \right] = \frac{\eta_{exec}(2^{k-1})}{\eta_{exec}(2^k)} T_{CmpExec}^{IM}.$$

Consistent with the assumptions given above,  $\eta_{exec}(2^k) < \eta_{exec}(2^{k-1}) \leq 2\eta_{exec}(2^k)$ . Thus, using either the  $\eta_{exec}(2^{k-1})$  information, if it is known, or the inequalities stated in the previous paragraph, the expected range for  $T_{CmpExec}^{FFS}$  is:

$$T_{CmpExec}^{TM} < T_{CmpExec}^{FFS} \le 2T_{CmpExec}^{TM} \le 2\eta_{exec}(2^k).$$

An execution-time estimate for the task redistribution recovery option is more difficult than for the previous options. Consider a task executing on a submachine of size  $2^k$  in MIMD mode. If a PE becomes faulty and its **subtasks** are distributed equally to the  $2^k - 1$ fault-free **PEs** in the submachine, the remaining execution time is bounded as follows:

$$T_{CmpExec}^{TR} \leq \frac{2^k}{2^k - 1} \left[ \eta_{exec}(2^k) - \tau \right] = \frac{2^k}{2^k - 1} T_{CmpExec}^{TM}.$$

Consider the situation where the faulty **PE's subtasks** cannot be distributed equally among the fault-free **PEs.** In the worst case, all the faulty **PE's subtasks** would be assigned to a single PE and the remaining execution time could be twice that of the remaining execution time on a fault-free submachine.

In general, it is expected that  $T_{CmpExec}^{TR} \leq T_{CmpExec}^{FFS}$  because the fault-free subdivision option can be thought of as a subset of the task redistribution option where the task is redistributed to half the **PEs** in the original submachine. Furthermore, it is expected that  $T_{CmpExec}^{TR} > T_{CmpExec}^{TM}$  based on the earlier assumption that the original submachine size was selected for minimum execution time. Therefore, the range on  $T_{CmpExec}^{TR}$  is given by:

$$T_{CmpExec}^{TM} < T_{CmpExec}^{TR} \le T_{CmpExec}^{FFS}$$

In cases where an equal distribution of the task load among the fault-free **PEs** is possible, the upper bound of  $T_{CmpExec}^{FFS}$  in the above inequality can be replaced by  $min((2^k/2^k - 1)T_{CmpExec}^{TM}, T_{CmpExec}^{FS})$ . By combining the results of the inequalities for remaining execution time determined in this subsection, the following ordering is established.

$$0 < T_{CmpExec}^{TM} < T_{CmpExec}^{TR} \le T_{CmpExec}^{FFS} \le 2\eta_{exec}(2^k)$$

Although the above inequality indicates that task migration is the best reconfiguration option when the decision is based on  $T_{CmpExec}$  being the dominant factor, it has already been shown that task migration is not the best option when considering  $T_{Plan}$  and/or  $T_{Trnsfr}$ . Therefore, there is no clear choice based on the analysis thus far.

# 7. Penalty for Wrong Choice

In the previous section, a quantitative framework was developed that attempts to relate various reconfiguration parameters. Some of the parameters can be predicted with good precision on real machines, while other parameters can only be coarsely bounded. The next step is to determine if a heuristic can be found that is based on the information available. In this section, a combination of probabilistic analysis and worst-case analysis is used to develop useful guidelines for choosing among reconfiguration options on real machines in practical situations.

Consider the relative magnitudes of  $\eta_{exec}(2^k)$ ,  $T_{Plan}$ , and  $T_{Trnsfr}$ . In general, for tasks with short execution times, it is better to restart the task when a PE becomes unusable rather than permanently and significantly increasing the execution time by including periodic checkpointing. Therefore, dynamic reconfiguration is generally not considered for tasks

unless the estimated execution time for the task,  $\eta_{exec}(2^k)$ , is orders of magnitude larger than  $T_{Trnsfr}$  and  $T_{Plan}$ .

One of the most common cumulative distribution functions assumed in reliability models is the exponential distribution,  $F(t) = 1 - e^{-\lambda t}$  [SiS82]. <u>*F(t)*</u> represents the probability that a PE fault will occur between time 0 and time t, inclusive. The parameter  $\lambda$  describes the rate at which failures occur in time.

The reliability function, <u>**R**(t)</u>, is defined as  $\mathbf{R}(t) = 1 - \mathbf{F}(t) = e^{-\lambda t}$ . For a parallel system submachine of size 2<sup>k</sup> PEs, where all the PEs must be operational for the submachine to be operational, the submachine reliability function, <u>**R**<sub>sm</sub>(t)</u>, is the product of the individual PE reliability functions.

$$R_{sm}(t) = \prod_{i=1}^{2^{k}} R(t) = \prod_{i=1}^{2^{k}} e^{-\lambda t} = e^{-2^{k}\lambda t}$$

Thus, the submachine-failure probability distribution function,  $\underline{F_{sm}(t)}$ , for a submachine of size  $2^k$  PEs is given by:

$$F_{sm}(t) = 1 - e^{-2^k \lambda t}.$$

Consider the conditional probability that a failure occurs at or before time  $.9\eta_{exec}(2^k)$  given that a failure occurs at or before time  $\eta_{exec}(2^k)$ .

$$Pr[t \le .9\eta_{exec}(2^{k}) | t \le \eta_{exec}(2^{k})] = \frac{Pr[t \le \eta_{exec}(2^{k}) | t \le .9\eta_{exec}(2^{k})]Pr[t \le .9\eta_{exec}(2^{k})]}{Pr[t \le \eta_{exec}(2^{k})]}$$
$$= \frac{1 - e^{-2^{k}\lambda.9\eta_{exec}(2^{k})}}{1 - e^{-2^{k}\lambda.9\eta_{exec}(2^{k})}}$$

This probability approaches 0.9 as  $\eta_{exec}(2^k)$  approaches zero from the positive direction, and it monotonically approaches 1 as  $\eta_{exec}(2^k)$  increases. Therefore, when  $\eta_{exec}(2^k)$  is 100 times greater than  $T_{Trnsfr}$ , there is a 0.9 or greater probability that a failure will occur by time  $90T_{Trnsfr}$ , given that a failure occurs by the time the program has completed. Thus, for this case, there is a high probability that  $\tau$ , the time the failure occurs, will be less than or equal to  $90T_{Trnsfr}$ , and  $T_{CmpExec} = \eta_{exec}(2^k) - \tau > 10T_{Trnsfr}$ . For the case where  $\eta_{exec}(2^k)$  is more than 100 times greater than  $T_{Trnsfr}$ , there is an even greater probability that  $\eta_{exec}(2^k) - \tau > T_{Trnsfr}$ . Thus, there is a high probability that  $T_{CompExec}$  will be much greater than  $T_{Trnsfr}$  when a fault occurs.

Consider the penalty of choosing the wrong reconfiguration option. The worst-case penalty,  $\underline{T_{Penalty}^{WC}}$ , is defined to be the worst-case difference between the expected completion time of a task after choosing a suboptimal reconfiguration option and the expected completion time of a task after choosing the optimal reconfiguration option. For example, if the task redistribution option was chosen, but the task migration option would have resulted in the earliest completion time for the task, the worst-case penalty would be:

 $T_{Penalty}^{WC} = \max \left( T_{Plan}^{TR} + T_{Trnsfr}^{TR} + T_{CmpExec}^{TR} \right) - \min \left( T_{Plan}^{TM} + T_{Trnsfr}^{TM} + T_{CmpExec}^{TM} \right),$ 

where the maximum and minimum refer to the ranges for the parameters. Here, two cases are considered: 1) the reconfiguration choice was made assuming that the remaining execution time was much greater than the time to transfer the task code and data  $((\eta_{exec}(2^k) - \tau) > T_{Trnsfr})$ , and 2) the reconfiguration choice was made assuming that the remaining execution time was much less than the time to transfer the task code and data  $((\eta_{exec}(2^k) - \tau) < T_{Trnsfr})$ . The case where  $(eta_{exec}(2^k) - \tau)$  and  $T_{Trnsfr}$  are of the same order is not of interest in a worst-case analysis.

First, consider the case where it was incorrectly assumed that  $(\eta_{exec}(2^k) - \tau) > T_{Trnsfr}$ . In this case, from the results of Subsection 4.4, the task migration option would have been chosen. If the fault-free subdivision option is the optimal one, the worst-case penalty would be:

$$T_{Penalty}^{WC} < \max(T_{Plan}^{TM} + T_{Trnsfr}^{TM}) - \min(T_{Plan}^{FFS} + T_{Trnsfr}^{FFS}),$$

because the best expected time to complete execution on a fault-free subdivision is greater than the expected time to complete execution after task migration. Furthermore, for machines like PASM and nCUBE 2,  $T_{Plan}^{TM} - T_{Plan}^{FFS}$  is negligible, and  $T_{Trnsfr}^{TM} - T_{Trnsfr}^{FFS}$  is generally on the order of hundreds of milliseconds (see Table II). Thus, in this case,  $T_{Penalty}^{WC}$  is on the order of  $T_{Trnsfr}^{TM}$  (recall  $T_{Trnsfr}^{FFS} = 0$ ).

If instead the task redistribution option is the optimal choice for this example, the worst-case penalty would be:

$$T_{Penalty}^{WC} < \max(T_{Plan}^{TM} + T_{Trnsfr}^{TM}) - \min(T_{Plan}^{TR} + T_{Trnsfr}^{TR}).$$

Again, the best expected time to complete execution after task redistribution is greater than the expected time to complete execution after task migration. For PASM and **nCUBE** 2,  $T_{Penalty}^{WC}$  is on the order of  $T_{Trnsfr}^{TM}$  (see Table II).

Now, consider the case where it was incorrectly assumed that  $(\eta_{exec} - 7) < < T_{Trnsfr}$ . In this situation, either the fault-free subdivision or task redistribution option would have been chosen. If the fault-free subdivision option was chosen when the task migration option would have been better (because in actuality  $(\eta_{exec} - \tau) > > T_{Trnsfr}$ ), the penalty for making the wrong choice is given by:

$$T_{Penalty}^{WC} = \max(T_{Plan}^{FFS} + T_{Trnsfr}^{FFS} + T_{CmpExec}^{FFS}) - \min(T_{Plan}^{TM} + T_{Trnsfr}^{TM} + T_{CmpExec}^{TM}).$$

Substituting values from the analysis in Subsection 6.6.4, results in:

$$T_{Penalty}^{WC} = \max(T_{Plan}^{FFS} + T_{Trnsfr}^{FFS}) - \min(T_{Plan}^{TM} + T_{Trnsfr}^{TM}) + (2T_{CmpExec}^{TM} - T_{CmpExec}^{TM})$$
$$= \max(T_{Plan}^{FFS} + T_{Trnsfr}^{FFS}) - \min(T_{Plan}^{TM} + T_{Trnsfr}^{TM}) + \eta_{exec}(2^{k}) - \tau.$$

Recall the value of  $\eta_{exec}(2^k)$  is assumed to be much larger than  $T_{Trnsfr}$  when reconfiguration options are to be considered. Thus, in the worst case, the penalty for incorrectly assuming  $(\eta_{exec} - 2) < \langle T_{Trnsfr}$  is much greater than incorrectly assuming  $(\eta_{exec} - 2) > T_{Trnsfr}$ . A similar analysis for the case where task redistribution was erroneously chosen over task migration results in the same potential for a large penalty.

To summarize this section, two conclusions are made: first, it is expected that there is a high probability that  $T_{CmpExec}$  will be much greater than  $T_{Trnsfr}$  when a fault occurs, and second, that the worst-case penalty for incorrectly assuming this is true is far less than the worst-case penalty for incorrectly assuming the opposite. Therefore, a mathematical justification for choosing a reconfiguration option by considering only the time required to complete the task has been established. Combining this result with the results of Subsection 4.4, the choice of reconfiguration strategy becomes one of choosing to migrate the task if an idle submachine exists. If this option is not available, the next best option is task redistribution. Finally, if the task does not lend itself to redistribution, a fault-free subdivision can be used to complete the task.

The model parameter value ranges established in Section 6.6 are in some cases very coarse, e.g.,  $T_{CmpExec}$  for tasks with data-dependent (nondeterministic) execution times, and therefore do not provide the information needed to determine the best reconfiguration option. However, the analysis in this section has made it possible to establish a good set of reconfiguration guidelines.

#### 8. Summary

A quantitative model of system reconfiguration due to a PE or PE memory module fault was examined. Four fault recovery options were discussed and the parameters required to determine their respective costs were identified. For the fault-free subdivision, task migration, and task redistribution recovery options (future work will add the remote memory recovery option), the model parameters were categorized into one of three categories: time to plan for the reconfiguration option ( $T_{Plan}$ ), time to move the task data and code ( $T_{Trnsfr}$ ), and time to complete task execution after reconfiguration ( $T_{CmpExec}$ ). The relative times for each reconfiguration option considered were examined for each category and the options were ranked when possible. Actual parameters collected on the PASM experimental prototype and the **nCUBE** 2 commercial machine were used to support the analysis.

For the system architectures considered,  $T_{Plan}$  is generally much smaller than  $T_{Trnsfr}$ . Furthermore, when basing the reconfiguration decision only on  $T_{Trnsfr}$  (ignoring  $T_{CmpExec}$ ), the fault-free subdivision or task redistribution options will result in the smallest total execution time for the task. The choice between the fault-free subdivision and task redistribution options will depend on the task being executed.

It was shown that for those tasks where dynamic reconfiguration should be considered, there is a high probability that the expected value of  $T_{CmpExec}$  will be greater than  $T_{Trnsfr}$ . Thus,  $T_{CmpExec}$  becomes the primary parameter to consider when choosing among reconfiguration options. When  $T_{CmpExec}$  is the dominant factor, the task migration option results in the earliest task completion. Task redistribution is the next best option. However, in the worst case, task redistribution can require as much time as completing the task on a fault-free subdivision.

Task execution times used in the model may be just expected values when execution times are data dependent and therefore nondeterministic. Because of this, a worst case analysis was performed. An examination of the penalty for choosing the wrong reconfiguration option provides further justification for basing the reconfiguration decision on  $T_{CmpExec}$ . An analysis of the worst-case penalties reveals that the penalty for assuming

 $T_{CmpExec}$  to be much greater than  $T_{Plan}$  and  $T_{Trnsfr}$  is much less than the penalty for assuming otherwise. Thus, using a quantitative framework, it has been shown that task migration is the best dynamic recovery option when  $T_{CmpExec}$  is expected to be much greater than  $T_{Plan}$  and  $T_{Trnsfr}$  and when the execution time of a task is nondeterministic.

*Acknowledgments:* The authors gratefully acknowledge useful discussions with James Armstrong, **Hasan** Cam, Peter Wahle, and Dan Watson.

# References

- [BaB91] V. Balasubramanian and P. Banerjee, ''CRAFT: Compiler-assisted algorithmbased fault tolerance in distributed memory multiprocessors,'' 1991 International Conference on Parallel Processing, Vol. I, August 1991, pp. 505-508.
- [Bat82] K. E. Batcher, "Bit-serial parallel processing systems," *IEEE Transactions on Computers*, Vol. C-31, No. 5, May 1982, pp. 377-384.
- [Bla90] T. Blank, 'The MasPar MP-1 architecture,'' *IEEE Compcon*, February 1990, pp. 20-24.
- [CrG85] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," 1985 International Conference on Parallel Processing, August 1985, pp. 531-540.
- [DaG85] F. Darema-Rodgers, D. A. George, V. A. Norton, and G. F. Pfister, *Environment and System Interface for VMIEPEX*, Research Report RC11381 (#51260), IBMT. J. Watson Research Center, 1985.
- [DaM85] A. T. Dahbura, G. M. Masson, and C. Yang, "Self-implicating structures for diagnosable systems," *IEEE Transactions on Computers*, Vol. C-33, No. 8, August 1985, pp. 718-723.
- [FiC91] S. A. Fineberg, T. L. Casavant, and H. J. Siegel, "Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting," *Journal of Parallel and Distributed Computing*, Vol. 11, No. 2, March 1991, pp. 239-251.
- [Fly66] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, Vol. 54, No. 12, December 1966, pp. 1901-1909.
- [FrT89] T. M. Frazier and Y. Tamir, "Application-transparent error-recovery techniques for multicomputers," Fourth Conference on Hypercubes, Concurrent Computers, and Applications, March 1989, pp. 103-108.

- [HaL87] J. Hastad, T. Leighton, and M. Newman, "Reconfiguring a hypercube in the presence of faults," 19th ACM Symposium on the Theory of Computing, 1987, pp. 274-284.
- [HaM86] J. P. Hayes, T. N. Mudge, Q. F. Stout, and S. Colley, "Architecture of a hypercube supercomputer," 1986 International Conference on Parallel Processing, August 1986, pp. 653-660.
- [Hun89] D. J. Hunt, ''AMT DAP a processor array in a workstation environment,'' *Computer Systems Science and Engineering*, Vol. 4, No. 2, April 1989, pp. 107-114.
- [Int85] Intel Corporation, *A New Direction in Scientific Computing*, Order # 28009-001, Intel Corporation, 1985.
- [KiN91] S. D. Kim, M. A. Nichols, and H. J. Siegel, "Modeling overlapped operation between the control unit and processing elements in an SIMD machine," *Journal* of Parallel and Distributed Computing, Special Issue on Modeling of Parallel Computers, Vol. 12, No. 4, August 1991, pp. 329-342.
- [KrM88] R. Krishnamurti and E. Ma, "The processor partitioning problem in specialpurpose partitionable systems," 1988 International Conference on Parallel Processing, Vol. I, August 1988, pp. 434-443.
- [Law75] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, Vol. C-24, No. 12, December 1975, pp. 1145-1155.
- [LeM88] D. Le Métayer, 'ACE: An Automatic Complexity Evaluator,' ACM Transactions on Programming Languages and Systems, Vol. 10, No. 2, April 1988, pp. 248-266.
- [LiM87] G. J. Lipovski, and M. Malek *Parallel Computing: Theory and Comparisons*, Wiley, New York, 1987.
- [LiS88] M. Livingston and Q. F. Stout, "Fault tolerance of allocation schemes in massively parallel computers," *Frontiers* '88: The Second Symposium on the Frontiers of Massively Parallel Computation, October 1988, pp. 491-494.
- [NaM92] W. G. Nation, A. A. Maciejewski, and H. J. Siegel, "Exploiting concurrency among tasks in partitionable parallel supercomputers," *Journal of Parallel and Distributed Computing*, Special Issue on Performance of Supercomputers, to appear, October 1993.
- [Ncu90] nCUBE Corporation, *nCUBE* 2 *Processor Manual*, Order # 101636, nCUBE Corporation, December 1990.
- [Nut77] G. J. Nutt, "Microprocessor implementation of a parallel processor," *Fourth Annual Symposium on Computer Architecture*, March 1977, pp. 147-152.

- [Pea77] M. C. Pease, III, "The indirect binary n-cube microprocessor array," *IEEE Transactions on Computers*, Vol. C-26, No. 5, September 1977, pp. 458-473.
- [PfB85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, ''The IBM Research Parallel Processor Prototype (RP3): introduction and architecture,'' *1985 International Conference on Parallel Processing*, August 1985, pp. 764-771.
- [SaS93] G. Saghi, H. J. Siegel, and J. L. Gray, "A performance prediction and mode of parallelism case study using cyclic reduction on three machine classes," Special Issue on Performance of Supercomputers, to appear, October 1993.
- [ScS88] T. Schwederski, H. J. Siegel, and T. L. Casavant, 'A model of task migration in partitionable parallel processing systems," *Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation*, October 1988, pp. 211-214.
- [ScS90] T. Schwederski, H. J. Siegel, and T. L. Casavant, "Optimizing task migration transfers using multistage cube network," *1990 International Conference on Parallel Processing*, Vol. I, August 1990, pp. 51-58.
- [SiA92] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer-visionrelated tasks onto reconfigurable parallel-processing systems," *Computer*, Special Issue on Parallel Processing for Computer Vision and Image Understanding, Vol. 25, No. 2, February 1992, pp. 54-63.
- [Sie80] H. J. Siegel, "The theory underlying the partitioning of permutation networks," *IEEE Transactions on Computers*, Vol. C-29, No. 9, September 1980, pp. 791-801.
- [Sie90] H. J. Siegel, Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies, Second Edition, McGraw-Hill, New York, NY, 1990.
- [SiS82] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Equipment Corp., Bedford, MA, 1982.
- [SiS87] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture*, D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, DC, 1987, pp. 387-407.
- [TaS84] Y. Tamir and C. H. Sequin, "Error recovery in multicomputers using global checkpoints," 1984 International Conference on Parallel Processing, August 1984, pp. 32-41.

- [TuR88] L. W. Tucker and G. G. Robertson, "Architecture and applications of the Connection Machine," *Computer*, Vol. 21, No. 8, August 1988, pp. 26-38.
- [UyR85] M. U. Uyar and A. P. Reeves, ''Fault reconfiguration in a distributed MIMD environment with a multistage network,'' 1985 International Conference on Parallel Processing, August 1985, pp. 798-806.
- [UyR88] M. U. Uyar and A. P. Reeves, "Dynamic fault reconfiguration in a meshconnected MIMD environment," *IEEE Transactions on Computers*, Vol. C-37, No. 10, October 1988, pp. 1191-1205.