On Reconfiguration Latency in Fault-Tolerant Systems[1]

Hagbae Kim and Kang G. Shin
Real-Time Computing Laboratory, Dept. of Elec. Eng. and Comput. Sci.
The University of Michigan, Ann Arbor, MI 48109–2122

Chuck Roark
Defense Systems & Electronics Group, Texas Instruments Incorporated
P.O. Box 660246, MS 3148, Dallas, TX 75266

Abstract

Digital computers embedded in critical applications such as flight controls should be equipped with appropriate fault-tolerance schemes to ensure their reliable and safe operation in the presence of component failures. System reconfiguration, which enhances reliability by dynamically using spatial redundancy, is generally the most time-consuming fault-/error- handling stage.

The *reconfiguration latency*, defined as the time taken for reconfiguring a system upon failure detection or mode change, depends on many parameters, including the size of application programs and data, the CPU and memory speed, built-in testing capabilities, the type (cold, warm, or hot) of spares to use, the system architecture, and the reconfiguration strategy used. In this paper, we classify the reconfiguration techniques into four types: reconfigurable duplication, reconfigurable N-Modular Redundancy (NMR), backup sparing, and graceful degradation. For each type of reconfiguration, we (i) evaluate the reconfiguration latency by using several parameters accounting for the aforementioned parameters, and (ii) determine if this type of reconfiguration can meet the application required latency.

Index Terms — Reconfiguration latency, dynamic redundancy, processor and task parameters, backup sparing, graceful degradation, cold, warm, and hot spares

[1]The work reported was supported in part by a Texas Instruments Grant, the Office of Naval Research under Grant N00014–91–J–1115 and by the NASA under Grant NAG–1–1120. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agencies.

I. Introduction

Safety-critical applications like flight controls require their controller computers to be equipped with fault-tolerance schemes to perform their intended function even in the case of component failures, thus meeting stringent reliability requirements. Fault-tolerance is generally achieved via time and/or spatial redundancy. System reconfiguration upon component failure(s) is a common form of dynamic use of spatial redundancy. Upon detection of a component failure, the system is reconfigured to prevent the faulty component from affecting system operation by disconnecting it from the rest of the system.

In a real-time application which requires quick computer responses, the *reconfiguration latency* — defined as the time required for disconnecting the faulty component, bringing in a replacement component and loading it with the application program and data — is a key to system reliability, because reconfiguration is generally the most time-consuming fault-/error- handling stage.

Although many researchers have proposed reconfiguration algorithms for various system architectures [1,2,6,8,12], little work has been done to evaluate the reconfiguration latency. In [5], the reconfiguration time was measured as the detection and isolation time via fault-injection experiments in the Fault-Tolerant Multiprocessor (FTMP), and the average reconfiguration time was experimentally measured to be 82 milliseconds on the FTMP [11], which is consumed for switching tasks and setting up interconnection. Roark *et al.* [9] reported the reconfiguration latency to range from 40–90 milliseconds for specific load-module sizes of several demonstration applications in a pooled-spares system implemented in the Dynamic Reconfiguration Demonstration System (DRDS) Program.[2] Although several aspects of reconfiguration affecting the reconfiguration latency were mentioned [3], the reconfiguration latency was modeled as a certain variable lying in a deterministic interval, as was done in [9].

In this paper we focus on the reconfiguration latency by analyzing the effects of all parameters associated with the reconfiguration process. We classify reconfiguration techniques into four types: reconfigurable duplication, reconfigurable N-Modular Redundancy (NMR), backup sparing, and graceful degradation. For each type of reconfiguration, we describe all the features of reconfiguration, which are generally composed of switching in power and bus connections, running power-up Built-In-Tests (BITs) on a spare, loading software programs and data from a permanent

[2] The DRDS Program is being developed to prove the feasibility of pooled spares for next generation weapon systems by Texas Instruments (TI) Incorporated under a contract from the Naval Air Warfare Center, Indianapolis, IN.

storage medium to the spare CPU and memory, and initializing software. First, we define several parameters that account for task size, CPU speed, the transfer rate of bus/interconnection, the number of interconnections (links) between a faulty module and its replacement module, and the types of spares. We evaluate qualitative and quantitative effects of these parameters on the reconfiguration latency for the four types of reconfiguration.

The paper is organized as follows. In Section II we specify task size, and processor & system capabilities by defining several parameters. The assumptions required to formalize our analysis are also presented there. In Section III we investigate the reconfiguration latency for each of the four types of reconfiguration as a function of the parameters defined in Section II. Section IV presents an example of evaluating and using the reconfiguration latency, especially for backup sparing. The paper concludes with Section V.

II. Preliminaries

When a failure is detected and its source is identified, the system (hardware and/or software) should be reconfigured to remove the failed component from active use. This process of changing the system organization or component interconnection may be invoked due to a mode change or a component failure, and it may occur during a non-operational period. However, throughout the paper we are primarily concerned with dynamic reconfiguration that enables the system to tolerate dynamically- and randomly- occurring faults during a mission. The time required for dynamic reconfiguration is critical to the operation of a real-time system, because reconfiguration — which is usually the most time-consuming fault-/error- handling stage — is the only means to remove any permanent fault and because after completing reconfiguration, the system must complete each control task within a certain time limit, called the *control system deadline* [4]. For the purpose of our analysis, we assume prompt and perfect fault detection and isolation. (Note that fault detection and isolation is itself an important topic and must precede the reconfiguration process.)

We define here several parameters that affect the reconfiguration latency. First, the application program or task determines the size of application code and data to be reloaded. (We assume that the core operating-system components are preloaded.) It is well-known that software download, if required, has the greatest impact on the reconfiguration latency. Let s_T be the task size measured in Kbytes. The reconfiguration latency may also depend on the speed of each individual processor, because the CPU speed greatly affects the program download time as well as the initialization time. Note that the time required for setting up the transfer, the operating system overhead, and the processing time for the transfer are intrinsically sensitive to the CPU speed. Let s_C be the CPU speed measured in MIPS. Bus/interconnection speed also influences the program download time, especially when this speed is slower than memory speed. This speed is determined by bus/interconnection bandwidth, network OS, and memory access time. Let s_B be the bus/interconnection speed measured in Mbytes/second.

In case of graceful degradation, task redistribution is required to transfer the tasks of a faulty module to the remaining active modules, where the transfer time depends upon the system architecture and the adopted reconfiguration algorithm. The reconfiguration algorithm decides which module(s) to take over the tasks of a faulty module. Let n_R be the number of interconnections between the faulty module and the module receiving the unfinished remaining tasks of the faulty module.

III. Evaluation Of Reconfiguration Latency

A majority of reconfiguration techniques are included in four classes of dynamic redundancy: (i) reconfigurable duplication, (ii) reconfigurable NMR, (iii) backup sparing, and (iv) graceful degradation. In this section, we derive the reconfiguration latency, denoted as t_{rl}, by investigating the effects of several parameters such as task size, processor capabilities, the system architecture, and the reconfiguration strategy for the four classes of dynamic reconfiguration.

A. Reconfigurable Duplication

A duplicated system is generally used to provide the capability of fault detection by comparing two modules' outputs.[3] When a fault is detected by a mismatch between two outputs, the duplicated system can be reconfigured by disconnecting the faulty module — identified by a diagnostic or test program, a watchdog timer, a self-checking circuit — and connecting a 'standby spare' running in parallel with the active module. As shown in Figure 1, a signal generated by a detected mismatch during comparison triggers reconfiguration through the control line.

Let t_s be the time needed for disconnecting the active module and then connecting the standby spare to the output line, and let t_I be the time (overhead) required for initializing hardware, a microprogram (to set the various control bits or registers), and the main application program, which enables the new module to have a smooth transition into full control of the system. Then, the reconfiguration latency is equal

[3] A module is defined as a logical block mapping a binary input vector to a binary output vector.

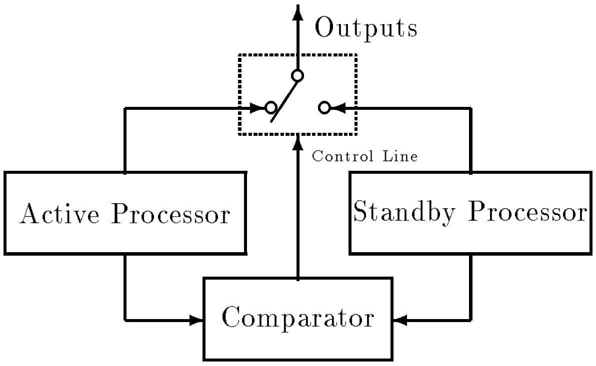

Figure 1: Structure of reconfigurable duplication.

to the time required for switching two modules and initializing the standby module:

$$t_{rl} = t_s + t_I, \tag{3.1}$$

where $t_{rl} = t_s$ if the standby module is always executing the same task for comparison-based fault detection. Since the standby spare is generally ready for performing the same tasks as the active module, the process of loading program and data is not necessary. Thus, the reconfiguration latency depends on the switching and/or initialization delay. Note that duplicated modules are generally located on the same bus and the bus interface unit in each module performs the switching function.

B. Reconfigurable NMR

The combination of N-modular redundancy and standby sparing, known as hybrid redundancy, is a promising approach to meeting the requirement of high reliability and availability. An NMR core is formed by connecting N identical modules to a majority voter, and several extra modules are provided as standby spares. The output of a faulty module differs from the majority-vote result, which is indicated by a disagreement detector. The reconfiguration process of this scheme corresponds to switching the faulty module with one of standby modules, which is also signaled by a disagreement detector.

Let t_s^{off} and t_s^{on} be, respectively, the time required for cutting off the faulty module and the time required for switching in the standby module and settling power-up transients, and let $t_s = t_s^{off} + t_s^{on}$. Let t_b and t_d be the time required for power-up BIT on the standby module and the time required for download and initialization on the spare module, respectively.

A switching strategy decides which modules to be switched in to replace the faulty module in the NMR core. In [10], two switching designs were proposed: (i) a *sequential* switch where all spares are ordered and the i-th spare is switched in to replace the i-th faulty module and (ii) a *rotary* switch where the spares arrange themselves in numerically increasing order of the voter positions and the lowest-numbered spare rotates to the highest voter position. The time spent for switching (t_s^{off} and t_s^{on}) depends on the adopted switching strategy and the switch complexity that intrinsically relies on the number of spares and the core size. The states of spares are also a key factor in determining the reconfiguration latency. A module in an unpowered state probably has a lower failure rate, and hence, one may keep the standby modules unpowered until they are switched in. In such a case, the reconfiguration latency significantly increases due to the time ($= t_b + t_d$) required to power up the selected spare and load or load & initialize the software (the application program and/or data of intermediate results) on the spare.

Considering all the parameters mentioned above, we can compute the reconfiguration latency — which depends on the switching strategy and the states of standby modules — as:

$$t_{rl} = t_s^{off} + t_s^{on} + t_b + t_d, \tag{3.2}$$

where t_b depends on the coverage of the BIT, the complexity of the spare, and BIT software (requiring a certain degree of hardware assistance). In Section C, we will examine the factors affecting t_d and derive t_d using the parameters introduced in Section II.

C. Backup Sparing

In addition to hybrid redundancy using the NMR core, the concept of backup sparing can be used in general multiprocessor structures like meshes and hypercubes. If any spare module can be used to replace any other working module, the spares are said to be "pooled". An active module periodically checkpoints its state on its backups so that a selected backup from the set of pooled spares may have state information to maintain consistency. In other words, the new active module restores the last checkpoint and re-executes all the operations that were executed by the previous active module since the checkpoint. The new active module can, then, start executing the remaining unfinished tasks and service new requests from the consistent state.

Let t_w be the time taken for selecting a spare to take over the remaining unfinished tasks of the faulty module. In case of dedicated spares which associate some spares locally with specific groups of active modules in order to minimize interconnection complexity, t_w can be made negligible by using well-controlled procedures. However,

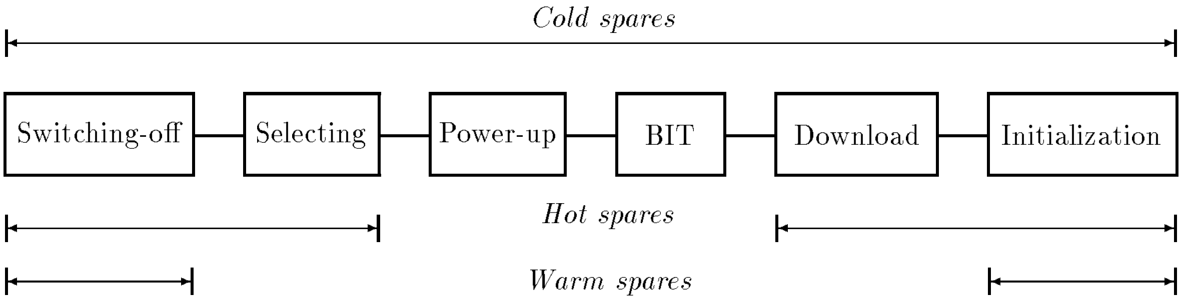

Figure 2: Reconfiguration steps of backup sparing

it could be considerable if a reconfiguration scheme is to be used for non-dedicated spares.

As shown in Figure 2, the reconfiguration process for this class is generally composed of (i) switching in power and bus connections, (ii) running BIT on the selected spare module, (iii) loading programs and data, (iv) initializing the software, some steps of which are not always needed depending on the state of on-line spares. When a spare to be switched in is determined upon fault detection and isolation, the selected spare is powered up and ready to become active. Since unpowered modules are likely to have a lower failure rate and the standby power requirements are lower than the active ones, unpowered spares are often kept in the form of *cold spares* despite their large time overhead to become active. Extensive testing is usually done during power-up before starting any normal operation, such as a comprehensive memory test. As mentioned in Section B, the time required for this power-up BIT (t_b) depends upon the complexity of the module, the accuracy of BIT, and the degree of hardware support for BIT software. In case cold spares are used, the reconfiguration latency is computed as:

$$t_{rl} = t_s^{off} + t_w + t_s^{on} + t_b + t_d. \tag{3.3}$$

While cold spares require the time to settle power-up transients and to run BIT ensuring that a healthy (nonfaulty) module is switched in the system, warm spares are kept powered up on-line ready to load and run the application software. Using Figure 2, we get the reconfiguration latency of warm spares:

$$t_{rl} = t_s^{off} + t_w + t_d. \tag{3.4}$$

Note that the term $t_s^{off} + t_w$ in the above expressions (3.3) and (3.4) needs to be replaced by $max\ \{t_s^{off},\, t_w\}$ if cutting off the faulty module and selecting a replacement module can be done in parallel.

The software program that is downloaded from a certain permanent storage to a spare CPU generally causes the largest delay. Let t_O, t_P, t_B, and t_I be the time required for setting up data transfer plus the operating system overhead, the processing time for the transfer, the transfer time on the bus/interconnection, and the time for initialization, respectively. Then, for a task of size s_T, we have:

$$t_d = t_O + t_I + s_T(t_B + t_P), \tag{3.5}$$

where t_O, t_P, and t_I are sensitive to the change of CPU speed, and t_B is sensitive to the change of bus/interconnection speed. If these parameters are measured to be $\{t_O^0, t_B^0, t_P^0, t_I^0\}$ for a certain protocol with s_C^0 and s_B^0, the estimated values of $\{t_O, t_B, t_P, t_I\}$ for different s_C and s_B are calculated by:

$$\begin{aligned} t_O &= c_C t_O^0, \quad t_P = c_C t_P^0, \\ t_B &= c_B t_B^0, \quad t_I = c_C t_I^0, \end{aligned} \tag{3.6}$$

where c_C and c_B are the coefficients indicating the degree of change in $\{t_O, t_I, t_P\}$ and t_B based on s_C and s_B, respectively. Although there are other factors affecting c_C and c_B (for example, c_C and c_B also depend on the inherent synchronization factors and the type of buffering, respectively), we consider only the effects of s_C and s_B because they have most predominant effects. c_C and c_B are inversely proportional to the module throughput (CPU speed s_C) and transfer rate of bus/interconnection (bus/interconnection speed s_B), respectively. (Note that c_C and c_B are decreased by using better s_C and s_B.) Hence, we get

$$c_C = \frac{s_C^0}{s_C} \quad \text{and} \quad c_B = \frac{s_B^0}{s_B}. \tag{3.7}$$

For example, from the actual measurements of a pooled-spares system implemented in Phase 1 of the DRDS Program in [7], i.e., 2.2 1750A Digital Avionics Information System (DAIS) MIPS and $s_B = 4.21$ $[Mbytes/second]$ (bus bandwidth 200 $[Mbytes/second]$ for 16 bit bus-width and memory access time 250 $[nanoseconds]$), we obtain:

$$\begin{aligned} t_O^0 + t_I^0 &= 29.8 \ [milliseconds], \\ t_B^0 &= 0.2375 \ [milliseconds/1Kbyte], \\ t_P^0 &= 0.3375 \ [milliseconds/1Kbyte], \end{aligned}$$

where $c_C = 2.2/s_C$ and $c_B = 4.21/s_B$.

By using Eqs. (3.5), (3.6), and (3.7), we estimate the value of t_d under the various conditions of s_T, s_C and s_B. When we change only the task size s_T while keeping s_C

and s_B constant, we obtain a linear equation for t_d:

$$t_d = As_T + B, \tag{3.8}$$

where

$$\begin{aligned} A &= t_B + t_P = c_B t_B^0 + c_C t_P^0 = \frac{s_B^0}{s_B} t_B^0 + \frac{s_C^0}{s_C} t_P^0, \\ B &= t_O + t_I = c_C(t_O^0 + t_I^0) = \frac{s_C^0}{s_C}(t_O^0 + t_I^0). \end{aligned}$$

When only the CPU speed s_C varies for fixed s_T and s_B, we obtain an equation containing two constants A and B for t_d:

$$t_d = \frac{A}{s_C} + B, \tag{3.9}$$

where

$$\begin{aligned} A &= s_C^0(t_O^0 + t_I^0 + s_T t_P^0), \\ B &= s_T t_B = s_T c_B t_B^0 = \frac{s_B^0}{s_B} s_T t_B^0. \end{aligned}$$

Likewise, for various s_B values we obtain t_d as:

$$t_d = \frac{A}{s_B} + B, \tag{3.10}$$

where

$$\begin{aligned} A &= s_T s_B^0 t_B^0, \\ B &= t_O + t_I + s_T t_P = c_C(t_O^0 + t_I^0 + s_T t_P^0) = \frac{s_C^0}{s_C}(t_O^0 + t_I^0 + s_T t_P^0). \end{aligned}$$

In case of hot spares that are always on-line executing the target software in parallel with the active hardware (dedicated hot spares) as in Section A, the reconfiguration latency reduces to:

$$t_{rl} = t_s^{off} + t_I, \tag{3.11}$$

which is suitable for applications requiring short latencies because all the steps but switching-off and initialization are not necessary for hot spares.

D. Graceful Degradation

When an error is detected and the faulty module is located in a multiprocessor system, the system is reconfigured to isolate the faulty module from the rest of the system. The faulty module may be replaced by a backup spare as discussed in Section C, or it may simply be switched off, thus degrading the system capability, i.e., *graceful degradation*. This technique uses redundant hardware as part of the normal operating resources at all times and allows the system performance to degrade gracefully while compensating for failures.

In a complex multiprocessor like a mesh or a hypercube, faulty modules are disconnected upon fault detection and identification, because a faulty module cannot be immediately repaired in many cases (nor replaced in the mode of graceful degradation). The remaining modules should be reconfigured into a small connected network and/or tasks are also redistributed by assigning the tasks of the failed modules to the remaining functional modules. For our analysis, we assume that each module can test its neighboring modules to determine their states (fault or fault-free), and the neighboring modules exchange predetermined test information and intermediate task results at regular intervals. The intermediate results of each module are stored in some of its neighbors and will be used by those neighbors for reconfiguration in case the module fails. We also assume that this procedure of testing and updating the intermediate results is synchronized throughout the system.

The reconfiguration latency is equal to the time spent for transferring and initializing the unfinished tasks of faulty modules. We assume that it takes t_w for a certain reconfiguration strategy to decide which modules to take over the tasks of the faulty modules, as has been done in various system architectures [1,2,6,8,12]. We define n_R as the number of interconnections between the faulty module and a module taking over the tasks of the faulty module. Let t_n be the time to transfer a unit of task (say 1 $[Kbyte]$) between the faulty module and its replacement module, which is called the *network latency* and depends upon the speed/bandwidth of an interconnection network and the distance (number of interconnections/links to go through). The network latency includes both the overhead to prepare for transferring a task in the source module (address generation, packetizing, etc.) and the overhead in the destination module induced due to acknowledgement, error check, and depacketizing. If the total size of the remaining unfinished tasks is s_T and the tasks are transferred in the block-data transfer mode, in which one unit of latency is required for a block of data/task elements, the reconfiguration latency is:

$$t_{rl} = t_s^{off} + t_w + t_n + s_T(t_B + t_P), \tag{3.12}$$

where

$$t_n = t_O + t_I + (n_R - 1)t_B,$$

and the effects of s_C and s_B upon $\{t_O, t_I, t_P, t_B\}$ were described in Section C. In the mode of single-data transfer where each data/task element requires one unit of

Table 1: t_d for various task sizes (s_T) [KBytes].

s_T	1	2	4	8	**16**	32	64	128
t_d	30.38	30.95	32.1	34.4	**39**	48.2	66.6	103.4

Table 2: t_d for various CPU speeds (s_C) [$MIPS$].

s_C	1	2	**2.2**	5	10	50	100	1000
c_C	2.2	1.1	**1**	0.44	0.22	0.044	0.022	0.0022
t_d	81.24	42.52	**39**	19.29	11.54	5.35	4.57	3.88

latency and the time to transfer it, the reconfiguration latency becomes:

$$t_{rl} = t_s^{off} + t_w + s_T(t_n + t_B + t_P). \tag{3.13}$$

IV. Example

In this section, we present an example of evaluating the reconfiguration latency for the demonstration system of [7] using *milliseconds* as the basic time unit. In the measurements of a pooled-spares system implemented in Phase 1 of the DRDS Program using 2.2 DAIS MIPS experimental system, the data of $\{t_O^0, t_B^0, t_P^0\}$ is given as $\{28.8, 0.2375, 0.3375\}$ with the condition of $s_T = 16$ [$Kbyte$], $s_C^0 = 2.2$ [$MIPS$], and $s_B^0 = 4.21$ [$Kbytes/millisecond$]. Suppose that $t_s^{off} = 1$, $t_w = 5$, $t_s^{on} = 80$, $t_b = 20$, and $t_I^0 = 1$ are observed, and $\{t_s^{off}, t_w, t_b, t_I^0\}$ are inversely proportional to the CPU speed s_C.

First, we begin with evaluating the time required for download and initialization t_d, which is most sensitive to task size and processor-capability parameters. Under the given condition, t_d is computed as $28.8 + 1 + 16 \times (0.2375 + 0.3375) = 39$. If we change s_T, s_C, or s_B, then t_d's are estimated using Eqs. (3.8), (3.9) and (3.10), as given in Tables 1, 2, and 3.

Table 3: t_d for various bus speeds (s_B) [$Kbytes/millisecond$].

s_B	2.105	**4.21**	8.42	16.84	33.68	42.1	67.36	84.2
c_B	2	**1**	0.5	0.25	0.125	0.1	0.0625	0.05
t_d	42.8	**39**	37.1	36.15	35.67	35.58	35.44	35.39

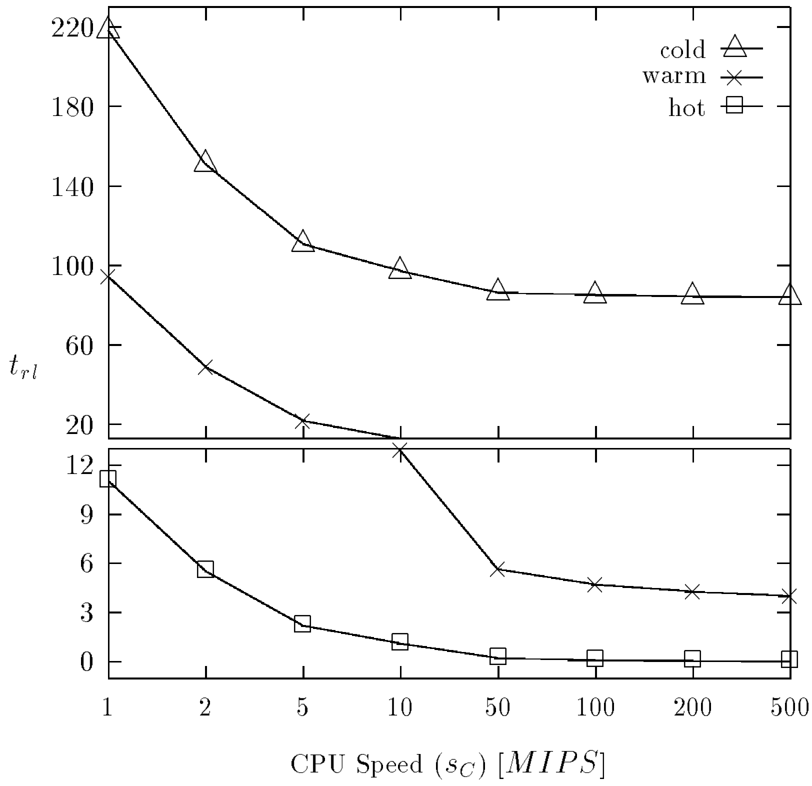

Figure 3: Reconfiguration latency vs. CPU speed for three types of spares: $s_B = 4.21$ [$Kbytes/milliseconds$].

Similarly, we derive t_{rl} under various conditions, i.e., type of spares, the CPU speed, and the bus/interconnection speed, with a fixed $s_T = 16$ [$Kbytes$]. Figure 3 plots the value of t_{rl} while varying s_C over three types of spares. Since most steps of system reconfiguration are sensitive to the CPU speed, t_{rl} is decreased significantly as s_C increases. However, t_{rl} of cold spares is not decreased below a certain value due to the insensitiveness of the time required for power-up transients to settle. The t_{rl} values of both warm spares and cold spares are not scaled directly with the CPU speed (but hot spares are directly scaled down) because the bus/interconnection transfer time t_B as well as t_s^{on} is independent of s_C. In Figure 4, we also plot t_{rl} while varying s_B. In this case, t_{rl} does not significantly change because the time required for only one step in system reconfiguration (t_B) relies on s_B.

Cold spares are generally useful for applications that require low fault rates of spares (faults occur more frequently in powered states) and do not have tight control system deadlines. In this type of spares, it takes longer to become operational due

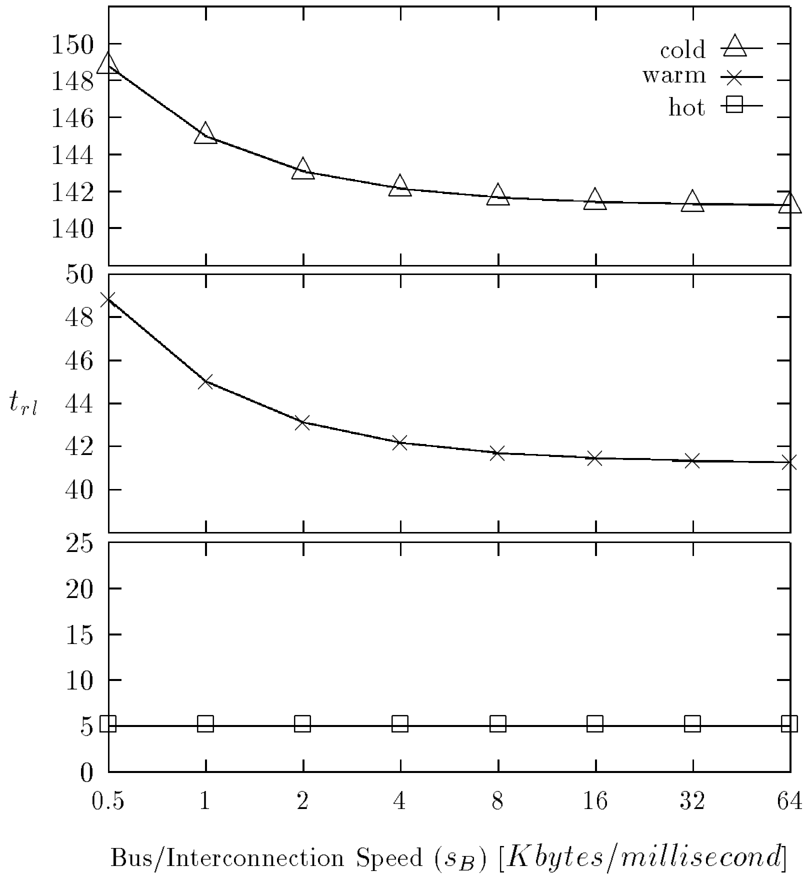

Figure 4: Reconfiguration latency vs. bus/interconnection speed for three types of spares: $s_C = 2.2$ [$MIPS$].

mainly to large t_s^{on} and t_b relative to the time required for other steps of reconfiguration. In Section III, we observed that the time required for most steps with cold (and warm) spares depends on the CPU speed. A fast CPU speed significantly decreases t_{rl} of cold or warm spares as shown in Figure 3, which allows cold or warm spares to be used effectively for meeting a certain system deadline. A higher transfer rate of bus/interconnection enhances the usefulness of cold or warm spares, as shown in Figure 4. However, only hot spares can satisfy stringent control system deadlines, requiring the reconfiguration latency of less than 1 [$millisecond$] in the example, for which s_C should be improved to have tens of MIPS for even hot spares.

V. Conclusion

We evaluated the reconfiguration latency of four classes of reconfiguration by using the time required for all reconfiguration steps. Specifically, we analyzed the effects of the task size, the CPU speed, and the bus/interconnection speed (and the number of links between a faulty module and its replacement module) upon the download and initialization time, which is a predominant contributor to the reconfiguration latency in backup sparing as well as graceful degradation. The reconfiguration latency can be decreased to meet a given control system deadline by using, for example, a fast CPU or a high transfer rate of bus/interconnection, in case of cold or warm spares. However, hot spares should be used to satisfy a tight control system deadline, because the time required for some steps of reconfiguration in using cold (or warm) spares is insensitive to these improved capabilities.

References

[1] P. Banerjee, "Strategies for reconfiguing hypercubes under faults," in *Proc. 2oth Annu. Int. Symp. on Fault–Tolerant Computing*, 1990.

[2] C. Chen, A. Feng, T. Kikuno, and K. Tori, "Reconfiguration algorithm for fault-tolerant arrays with minimum number of dangerous processors," in *Proc. 21st Annu. Int. Symp. on Fault–Tolerant Computing*, 1991.

[3] H. Kim and K. G. Shin, "Evaluation of fault-tolerance latency from real-time application's perspectives," Technical Report CSE-TR-201-94, CSE Division, EECS Department, The University of Michigan, 1994.

[4] H. Kim and K. G. Shin, "On the maximum feedback delay in a linear/nonlinear control system with input disturbances caused by controller-computer failures," *IEEE Trans. on Control Systems Technology*, vol. 2, no. 2, pp. 110–122, June 1994.

[5] J. H. Lala, "Fault detection, isolation and configuration in FTMP: Methods and experimental results," in *Proc. 5th IEEE/AIAA Digital Avionics Systems Conf.*, pp. 21.3.1–21.3.9, 1983.

[6] Y. H. Lee and K. G. Shin, "Optimal reconfiguration strategy for a degradable multi-module computing system," *Journal of the ACM.*, vol. 34, pp. 326–348, April 1987.

[7] D. Paul, C. Roark, and D. Struble, "Technical report on phase one of the dynamic reconfiguration demonstration system program," Technical report, Texas Instruments, Inc. NAWC-DRDS-P1-TR-0003, April 1992.

[8] C. V. Ramamoorthy and Y. W. Eva Ma, "Optimal reconfiguration strategies for reconfigurable systems with no repair," *IEEE Trans. on Computers*, vol. C-35, no. 3, pp. 278–280, March 1986.

[9] C. Roark, D. Paul, D. Struble, D. Kohalmi, and J. Newport, "Pooled spares and dynamic reconfiguration," in *Proceedings of NAECON'93*, pp. 173–179, May 1993.

[10] D. P. Siewiorek and E. J. McCluskey, "Switch complexity in systems with hybrid redundancy," *IEEE Trans. on Computers*, vol. C–22, no. 3, pp. 276–283, March 1973.

[11] T. B. Smith III and J. H. Lala, "Development and evaulation of a fault-tolerant multiprocessor (FTMP) computer. Volume IV: FTMP executive summary. NASA Contract Rep.172286," Technical report, NASA Langley Research Center, Langley, Va, February 1984.

[12] M. Uyar and A. Reeves, "Dynamic fault reconfiguration in a mesh-connected MIMD environment," *IEEE Trans. on Computers*, vol. 37, no. 10, pp. 1191–1205, October 1988.