.

Ordered Partial Task Compaction on Mesh Connected Computers

Oliver Diessel and Hossam ElGindy

Technical Report 96–11
September, 1996

Department of Computer Science and Software Engineering
The University of Newcastle, Callaghan NSW 2308, Australia
{odiessel,hossam}@cs.newcastle.edu.au

Abstract

Task compaction has been examined as a means of reducing fragmentation in partitionable machines based on multi–stage, hypercube, mesh and linear array interconnection networks. Ordered partial task compaction involves moving a subset of executing tasks without permuting their relative order to accommodate a request for a large submesh that would otherwise be blocked from entering the system. In this paper we develop the algorithms needed to find good allocation sites and to perform one–dimensional ordered partial compactions simply on mesh of processor and reconfigurable mesh architectures. We find that significant performance gains can be obtained in heavily loaded systems even when link delays are large.

Keywords: centralized control, cost and performance measures, partitionable architectures, reconfigurable networks, simulation, task allocation, task compaction.

1 Introduction

The partitionable multiple–SIMD/MIMD model allows compute processors to be shared among multiple, independently controlled tasks [10, 12, 3]. Such systems make effective use of the compute processors by adjusting the sizes of processor partitions to the sizes of the tasks, thereby allowing the number of tasks executed in parallel to be increased. Several contiguous processor allocation schemes have been proposed to manage the partitioning and allocation of contiguous blocks of processors under this model (see, for example, [6, 14]). Contiguous allocation schemes suffer from fragmentation of the available processors as variously sized tasks are allocated and deallocated. Tasks end up waiting in a queue to enter the system despite there being sufficient, albeit non–contiguous free processors available to service them. The time to complete a set of tasks is consequently longer, and the utilization of the compute resource is lower than it could be. When a task is blocked from entering the system because the available processors are fragmented, partial task compaction moves a subset of the allocated tasks to combine the free fragments between them if doing so allows the task to enter the system sooner.

The use of task migration to reduce fragmentation in partitionable multiple–SIMD machines was first investigated for multi–stage interconnection networks as part of the PASM project [8, 9]. Several full and partial task compaction methods were subsequently proposed for MIMD hypercubes, with efforts directed at devising optimal edge–disjoint migration algorithms (see, for example, [5, 2]). Results for the mesh architecture were reported in [13]. The use of reconfigurable buses and ordered partial task compaction were investigated for partitionable linear array machines in [4]. Significant performance improvements were found to be possible with task sets derived from field–collected traces and small link delays (1 second).

In this paper we extend the investigation of the use of reconfigurable buses to perform partial task compaction on two–dimensional reconfigurable meshes. We consider the problem of allocating a task by partial compaction to a reconfigurable mesh executing n tasks when it is not possible to allocate the task by other means. We describe methods by which the potential allocation sites can be found in $O(n^2)$ time and by which each of them can be assessed for feasibility and cost in $O(n)$ time. We propose three simple compaction scheduling algorithms — one for the mesh of processors architecture, and two for the reconfigurable mesh. We report on the experimental evaluation of their performance for a range of link transfer costs and discuss the significance of the results.

In the following section we describe the architecture, our assumptions, and our notation in detail. We then describe our ordered partial task compaction method before presenting and analyzing algorithms to perform the task compaction efficiently in Section 3. Results on the use of task compaction to improve the performance of reconfigurable meshes that employ *bottom–left* processor allocation and *first–come, first–served* scheduling are then reported in Section 4. Our findings are discussed in Section 5. We conclude with Section 6, which also mentions areas for future research.

2 Model

We consider a partitionable multi–SIMD reconfigurable mesh of compute processors, in which arbitrarily sized blocks of contiguous processors are independently controlled to operate in SIMD mode. An overview of this model is given in Figure 1. The compute resource consists of n interconnected processing elements (PEs) that are controlled by a set of m control processors (CPs) under the global control of a host. The host orchestrates the operation of the CPs, each of which broadcasts instructions for the PEs under its control over the CP–PE interconnection network.

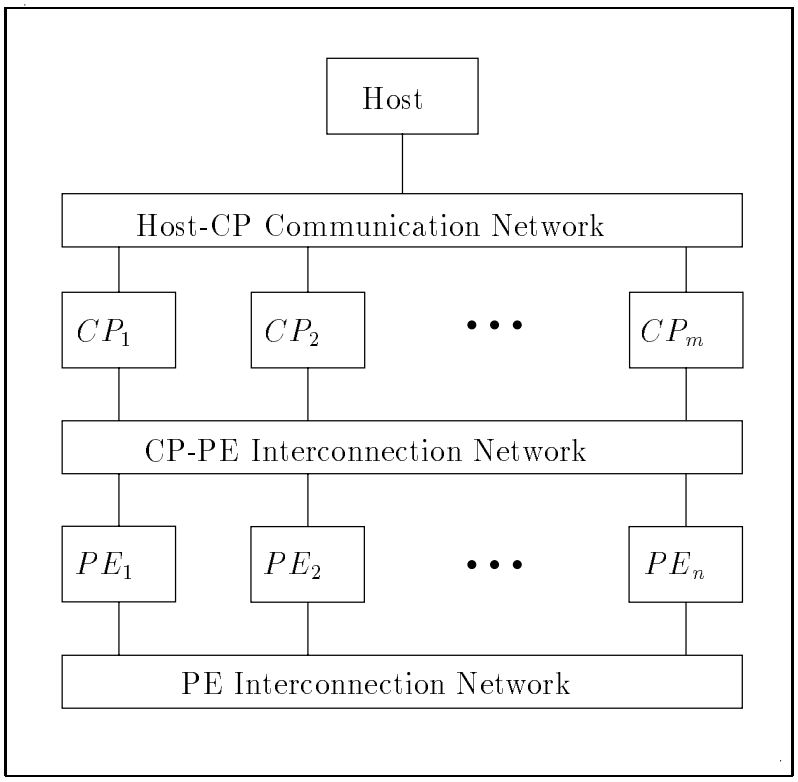

Figure 1: The multi–SIMD model of parallel computation.

In our model, the PEs are interconnected by a reconfigurable mesh of size $M(H, W)$, that consists of H rows and W columns of PEs arranged in a grid, as in Figure 2. Each processor is connected to its immediate neighbours to the north, south, east and west, when present, and has four similarly labeled I/O ports through which it can communicate with its neighbours. Each PE has control over a local set of short–circuit switches that allow the four I/O ports to be connected together in any combination. The 15 possible connection configurations are depicted in Figure 3. The PEs operate synchronously, in one machine cycle performing an arithmetic, logic or control operation, setting a connection configuration, and sending (receiving) a datum to (from) each I/O port. Processors are numbered from $P_{1,1}$ in the bottom–left corner, to $P_{H,W}$ in the top–right corner.

When a connection is set, signals received by a port are simultaneously available to any port connected to it. For example, if processors connect their northern and southern I/O ports by closing the appropriate switches as in the configuration (NS,E,W), data "broadcast"

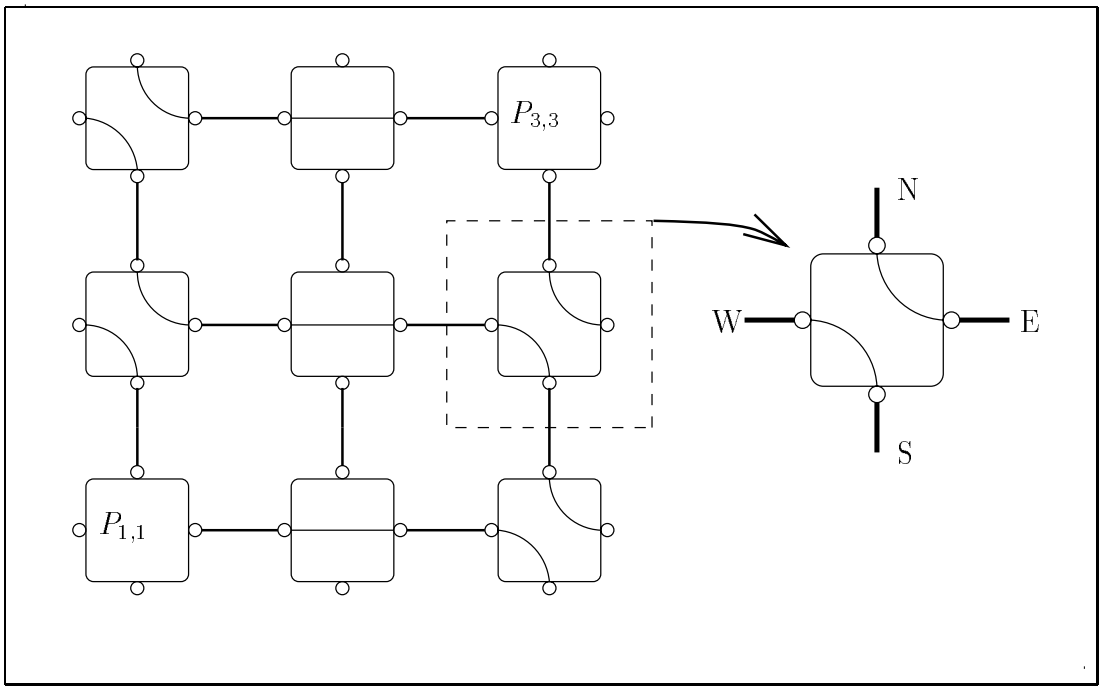

Figure 2: A reconfigurable mesh of size 3×3.

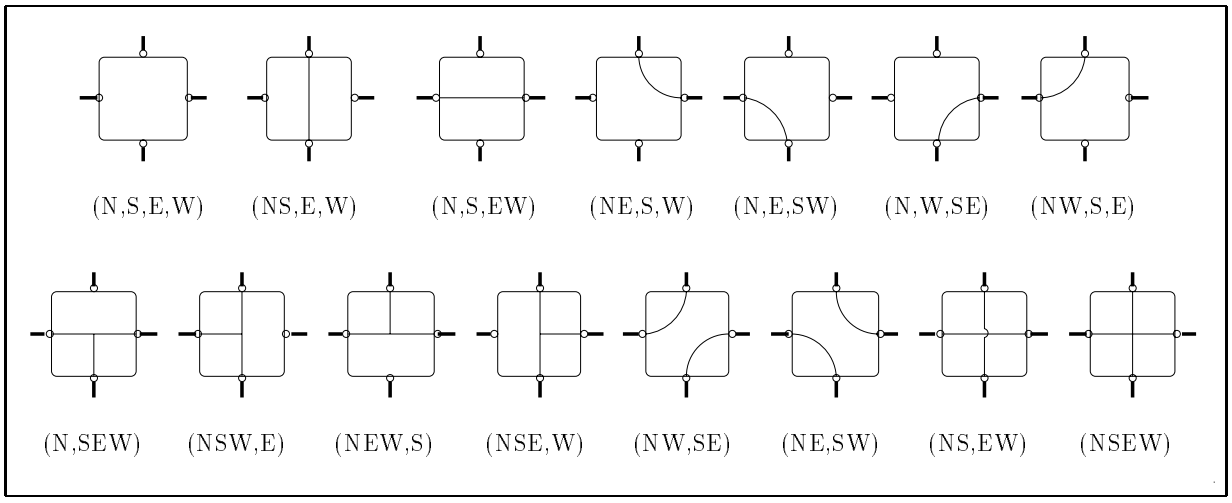

Figure 3: Reconfigurable mesh connection configurations.

onto the "column bus" can be read by all of the processors in a column. The model allows concurrent reading from a bus, requires exclusive writing to a bus, and usually assumes a constant time communication delay on arbitrarily large connected bus components. In Section 4.2 we investigate the effect of varying the communication latency over a range of reasonable values.

Moving a task involves switching the task out of context and setting the mesh to form buses to move the task elements (processor contexts and local memories). The source (target) processors of a move then write (read) their task elements to (from) their I/O ports. When all the elements of a task have reached their destination, the task is switched back into context. Compute processors have storage for two task elements between which they can switch in a single step. Storage for a task element that is switched out of context may be accessed from the compute processor's I/O port in a single step while the processor participates in the execution of another task. Compute processors participate in the execution of at most one task at a time, and all of the elements of a task need to be switched in for the task to be able to execute.

Overall management of tasks is accomplished in the following way: Tasks are queued by a sequential host as they arrive. A task allocator, executing on the host, attempts to find a location for the next pending task. If some executing tasks need to be compacted to accommodate the task, then a schedule for switching those tasks out of context, and moving their task elements is computed by the allocator. The allocator coordinates the switching of the reconfigurable mesh, task processors and task controllers according to the compaction schedule, and associates an available control processor with the new task and its allocation. If a location for the next pending task cannot be found, the task waits until one becomes available following one or more deallocations as tasks complete processing.

The following notation is used in this paper: A task $T_i(s_i, b_i)$ of size $s_i = (r_i, c_i)$ and base $b_i = P_{y_i,x_i}$ is allocated to a submesh of r_i rows and c_i columns of processors with bottom–leftmost processor P_{y_i,x_i}, $1 \leq y_i, 1 \leq x_i$ and top–rightmost processor $P_{y_i+r_i-1,x_i+c_i-1}$ with $y_i + r_i - 1 \leq H$ and $x_i + c_i - 1 \leq W$. The task T_i is said to be based at b_i. We denote the i^{th} row of processors R_i and the j^{th} column of processors C_j. The intersection of R_i and C_j is the processor $P_{i,j}$. The interval of processors $P_{i,k}, P_{i,k+1}, \ldots, P_{i,k+m}$ in the i^{th} row is denoted $R_i[k, k+m]$. A similar definition applies to $C_j[l, l+n]$. The intervals $R_i[k, k+m]$ and $C_j[l, l+n]$ intersect at processor $P_{i,j}$ iff $l \leq i \leq l+n$ and $k \leq j \leq k+m$. This notation is illustrated in Figure 4.

3 Algorithms

3.1 Bottom–Left Task Allocation

Bottom–Left allocation identifies the bottom–leftmost allocation site for an incoming task. In this study we adopt a method described by Zhu for finding the site [14]. Zhu's approach is to set a bit array corresponding to the tasks allocated to the mesh, and disallowed locations for the base of an incoming task. The array is then scanned row by row from left to right commencing with the bottommost row. The first clear bit found is the bottom–leftmost possible allocation site for the base of the request. While the technique is simple and guaranteed to find a location when one is available, it requires $\Theta(H \times W)$ time.

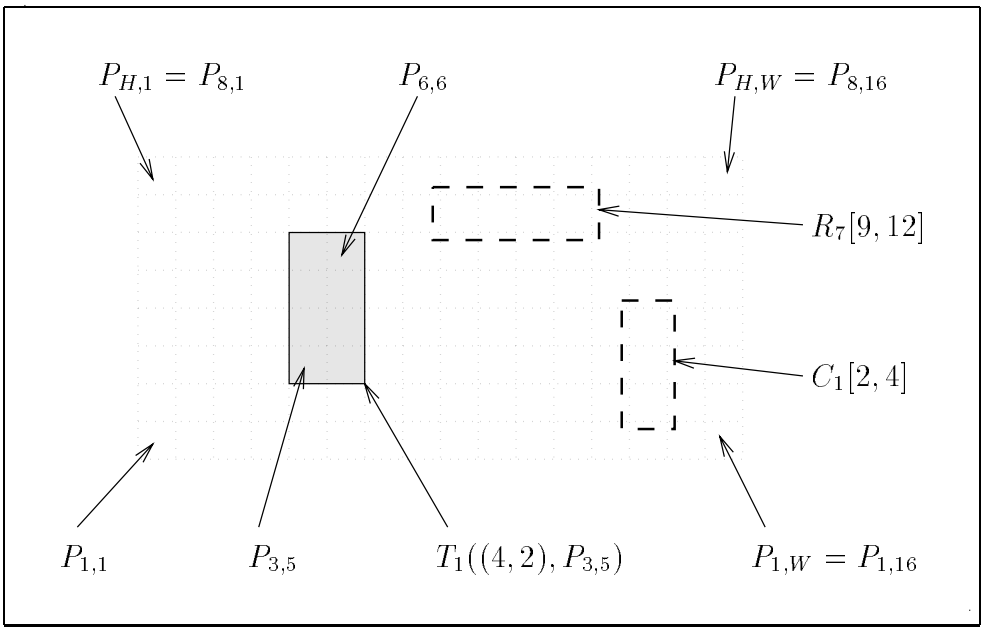

Figure 4: Notation.

The Bottom–Left approach to allocation is free of internal fragmentation, that is, it allocates precisely as many processors as requested, although like all contiguous allocation schemes, it suffers from external fragmentation over time as tasks are allocated and deallocated. External fragmentation is a problem when it prevents a request from entering the system because the free processors are not available in a sufficiently large contiguous block. In this study, we examine the benefits and costs of partially compacting the tasks allocated to the mesh in order to allocate tasks that would otherwise be blocked from entering the system due to external fragmentation.

3.2 Task Compaction

Task compaction is the process of squeezing tasks together so as to combine the interstitial free processors. When compaction is used to reduce fragmentation, these combined processors may then be allocated to the incoming task. Compactions are classified as either *full* or *partial*, depending on whether all allocated tasks are moved, or just a subset of them. In a planar architecture, such as a mesh, we can identify the class of *linear* compactions that move each task in a single direction only. When a compaction is linear, it may be *order–permuting* or *order–preserving* according to whether the relative order of tasks in each direction is altered or not. In this work we extend our investigation of *one–way one–dimensional order–preserving partial compactions* from linear reconfigurable arrays to two–dimensional reconfigurable meshes [4]. On a mesh, a one–way one–dimensional order–preserving compaction has the effect of sliding the set of tasks to be compacted in a single direction along a single dimension while preserving their relative order. Without loss of generality, we describe compacting the tasks to the right along the rows of the mesh. In VLSI circuit compaction this technique has been called "ploughing", which is a graphic term for describing the effect [7]. However, the process of moving the tasks according to a compaction schedule need not necessarily proceed in a single sweep. For the remainder

of this paper we use the term *compaction* to refer to one–way, one–dimensional order–preserving partial compaction unless we qualify the type of compaction we wish to refer to. Figure 5 contains an example of a compaction.

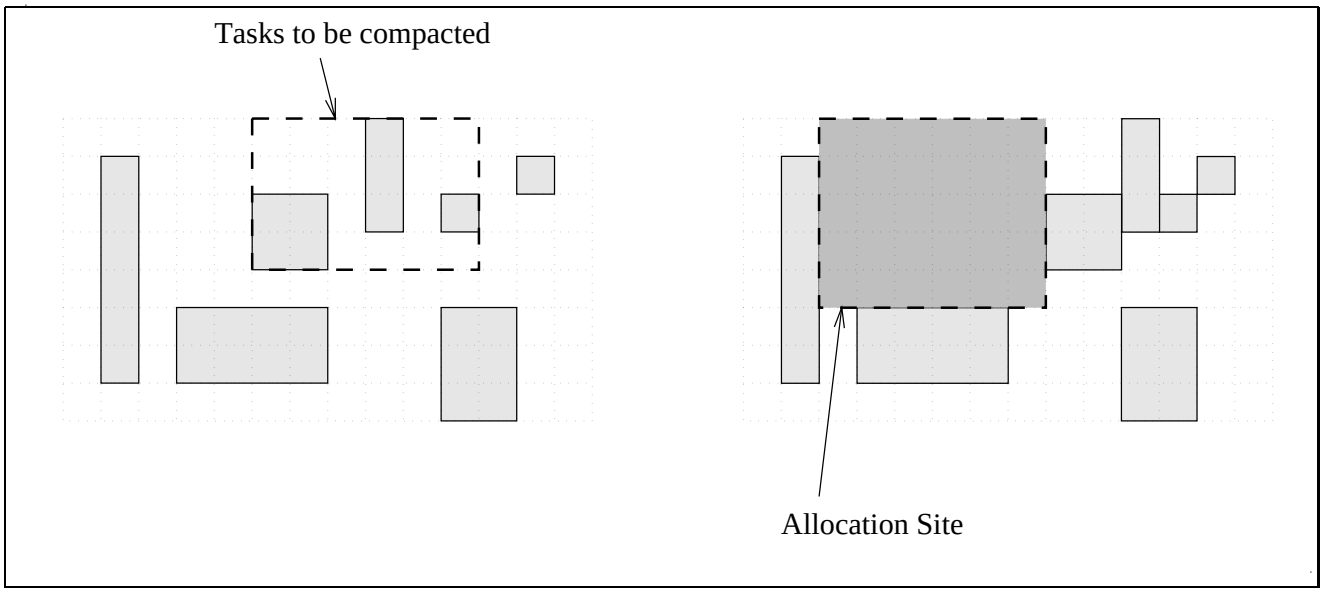

Figure 5: An example of a partial compaction. The initial arrangement on the left shows the tasks to be compacted so as to allocate a task of size 5×6. The final arrangement on the right indicates the allocation site.

3.2.1 Compaction Goals

We are interested in allocating a task T_{n+1} of size $s_{n+1} = (r_{n+1}, c_{n+1})$ to a reconfigurable mesh $M(H, W)$ that is executing the tasks $T_i(s_i, b_i), 1 \leq i \leq n$ when it is not possible to allocate T_{n+1} without compaction. Two subproblems arise naturally:

1. how to identify a good *allocation site*, a submesh of size s_{n+1}, efficiently, and
2. how to schedule the compaction so as:
   (a) to free the allocation site as quickly as possible,
   (b) to delay the tasks to be compacted as little as possible, and
   (c) to complete compacting the tasks as quickly as possible.

A "good" allocation site is one that facilitates the compaction goals. Central to satisfying these goals is the cost of moving a task. We address this issue next.

3.2.2 Cost of Moving a Task

On a mesh, a task element can only be moved over the links that connect neighbouring processors. To move a task it is switched out of context, processors then send the context they hold to their mesh neighbours until the task reaches its destination where it is switched back into context again. The time to move a task is proportional to the distance the task

has to move. The actual time spent needs to take into account the time to switch processes out of and into context, the size of a process, and the bandwidth of the mesh link. However, the task switch time for SIMD tasks is conceivably small compared with the link transfer time.

On a reconfigurable mesh, a task element can make use of reconfigurable buses to move to a non–local destination in constant time.

Lemma 1 *On a reconfigurable mesh, a task of size $s = (r, c)$ can be moved d processors to the right in $\Theta(\min(c, d))$ communication cycles, which is optimal.*

Proof: When $c \leq d$ the task elements of each row of the task must be moved past the right edge of the task. At least c cycles are needed to transfer these elements. The elements of a row can be moved to their destinations one after another in c cycles by forming buses that connect each source with its destination. All rows can be moved in parallel. When $c > d$, d task elements per row need to move past the right edge of the task. One element can move on a bus crossing the edge per cycle. Moving all elements of a row spaced d apart in parallel along buses of length d, all elements can be moved in d cycles. ∎

3.2.3 Identifying Potential Allocation Sites

A request of size $s_{n+1} = (r_{n+1}, c_{n+1})$ may not be satisfiable, even with compaction, although the number of free processors exceeds $r_{n+1} \times c_{n+1}$ because the system may be fragmented in such a way so as to prevent recombination of unallocated processors into a sufficiently large block of contiguous processors. (Consider, for example, a request of size $s = (\lfloor H/2 \rfloor + 1, \lfloor W/2 \rfloor + 1)$ to a mesh $M(H, W), H, W > 4$ executing a single task of that size already.) Our first challenge is to find a quick method for determining whether a task can be allocated with compaction or not. In this subsection we show that it is possible to reduce the number of potential allocation sites from $\Theta(H \times W)$ to $O(n^2)$, which is a considerable saving when the number of tasks is relatively small. Thereafter, we describe the construction of a direct dominance graph over the executing tasks that allows us to determine the feasibility of freeing the executing tasks from each candidate site in $O(n)$ time. In the worst case, we therefore spend $O(n^3)$ time determining whether the incoming task can be allocated with compaction.

Definition 1 *For the incoming task T_{n+1} of size $s_{n+1} = (r_{n+1}, c_{n+1})$ and the executing tasks $T_i(s_i, b_i), 1 \leq i \leq n$ with $s_i = (r_i, c_i)$ and $b_i = P_{y_i, x_i}$, we define a* top processor interval *for each executing task T_i, consisting of the interval of processors abutting its top edge, extending from its right edge to the left for a total length equal to one less than the combined widths of T_i and T_{n+1}, or until the left edge of the mesh is encountered. The leftmost processor of this interval is the base of T_{n+1} were its rightmost column placed on top of the leftmost column of T_i. The* top processor intervals *is defined to be the set $\{R_{r_i+y_i}[\max(1, c_i - c_{n+1} + 1), \min(c_i + x_i - 1, W - c_{n+1} + 1)] : 1 \leq i \leq n, r_i + y_i \leq H - r_{n+1} + 1\} \cup R_1[1, W - c_{n+1} + 1]$.*

Similarly, we define for each executing task T_i a right processor interval, *consisting of the the interval of processors abutting its right edge, extending from its top edge downward for a total length equal to one less than the combined lengths of T_i and T_{n+1}, or until the*

bottom edge of the mesh is encountered. The right processor intervals *is defined to be the set* $\{C_{c_i+x_i}[\max(1, r_i - r_{n+1} + 1), \min(r_i + y_i - 1, H - r_{n+1} + 1)] : 1 \leq i \leq n, c_i + x_i \leq W - c_{n+1} + 1\} \cup C_1[1, H - r_{n+1} + 1]$.

These intervals define the minimum cost locations for placing the base of the incoming task T_{n+1} if it is to be allocated in the neighbourhood of T_i. The definitions are illustrated in Figure 6.

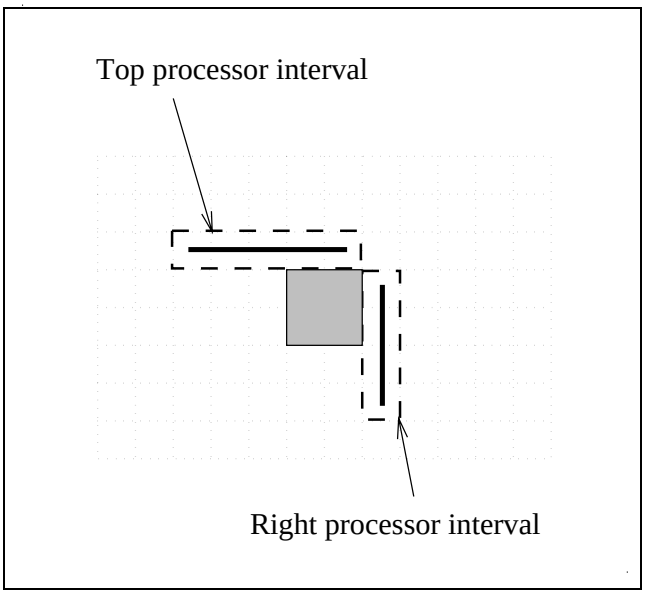

Figure 6: Definition of a top and right processor interval for a task of size 2×2 and an incoming task of size 3×4.

In the proof of the following theorem, we show that the set of processors at the intersection of the top and right processor intervals, which we denote $\mathcal{B}$, consists of the potential allocation site bases that cost least to free of executing tasks. For the example of Figure 5, the set is shown in Figure 7.

Theorem 1 *If* T_{n+1} *can be allocated by means of compaction, then the cost of freeing the executing tasks is minimized for an allocation site based at some processor in* $\mathcal{B}$.

Proof: The proof considers the time needed to free the space for the incoming task for all possible base positions as it is shifted along a row from the left edge of the mesh to the right. A similar argument applies to shifting the task up a column from the bottom edge of the mesh.

Let us consider the allocation site based at $P_{r,1}, 1 \leq r \leq H - r_{n+1} + 1$. Executing tasks lying within the allocation site are to be compacted to the right. The maximum number of allocated processors found on any row within the allocation site places a lower bound on the time needed to free the site by horizontal movements only. Assume the leftmost allocated processor(s) within the allocation site are in column C_c. As the base of the allocation site is shifted to the right from $P_{r,1}$ to $P_{r,c}$, additional allocated processors potentially become covered by the right edge of the allocation site, thereby increasing the time to free the site

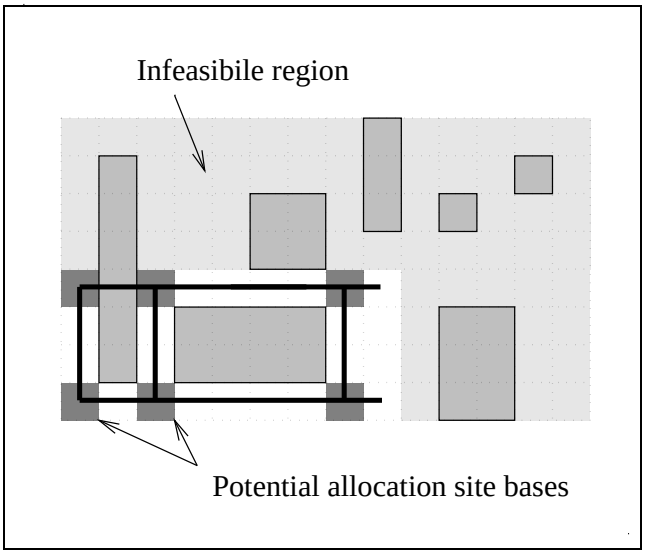

Figure 7: Potential allocation site bases for an incoming task of size 5×6.

of occupying tasks. However, it is not until the first task occupying the allocation site, T_l say, is completely uncovered by the left edge of the allocation site that the time needed to free the site of occupying tasks potentially decreases, since while any processors of T_l remain within the allocation site T_l must move to the right of it. Thus it is only necessary to check allocation sites based in the columns of processors $C_{x_i+c_i}$ immediately to the right of executing tasks T_i. The base is constrained from moving to either side with the potential of reducing the cost to free the allocation site by the presence of T_i to the left, and the possibility of covering additional tasks to the right. However, the columns $C_{x_i+c_i}$ need only be checked over the interval in which the task T_i potentially intersects the allocation site, namely $C_{x_i+c_i}[\max(1, r_i - r_{n+1} + 1), \min(r_i + y_i - 1, H - r_{n+1} + 1)]$.

By a similar argument it follows that it is only necessary to check sites in the rows of processors $R_{y_i+r_i}$ immediately above executing tasks T_i. These rows $R_{y_i+r_i}$ need only be checked in the interval in which the task T_i intersects the allocation site, $R_{y_i+r_i}[\max(1, c_i - c_{n+1} + 1), \min(c_i + x_i - 1, W - c_{n+1} + 1)]$.

Consider the top processor interval associated with a task T_i. The allocation site is constrained from moving below or above it without potentially increasing the cost to free the site. Allocation sites based to the right or left of the interval are potentially more costly than sites within the chosen column that intersect other top processor intervals. A similar argument applies to a right processor interval. Therefore if we consider potential bases within the top interval, the cost to free the allocation site is least where it intersects right intervals. These intersections are guaranteed to exist due to the fact that T_{n+1} cannot be allocated without compaction. ■

Note that the bottom and left edges of the mesh play a similar role to the top and right edges of a task, and therefore have top and right processor intervals associated with them. Since it is not possible for the base of the incoming task to be located in rows above $R_{H-r_{n+1}+1}$ and columns to the right of $C_{W-c_{n+1}+1}$, processor intervals within these regions are excluded from consideration.

Constructing the set $\mathcal{B}$ of potential bases for the incoming task requires $O(n^2)$ time if each member of the set of right processor intervals is used to check for intersections against each member of the set of top processor intervals. Since $O(n^2)$ potential base locations have to be identified, this is optimal in the worst case.

Allocation sites based at processors in $\mathcal{B}$ are not guaranteed to be feasible, since it may not be possible to compact the executing tasks within the allocation site to the right due to a lack of free processors. Determining the feasibility of the site is addressed next.

3.2.4 Assessing Allocation Site Feasibility

In order to determine whether the executing tasks within an allocation site can be moved out of the site by a right ordered compaction, we need to know whether sufficient free space exists to move the tasks within the allocation site beyond the site by compaction. With $O(n^2)$ sites to search, we need an efficient means of answering this question. Our solution is to build a direct dominance graph of the executing tasks, that also contains quantitative information about the arrangement of tasks, in order for us to determine the feasibility of a site in $O(n)$ time.

Definition 2 *(After [11]) Task V* dominates *a task T if, for some processor P_{r_V,c_V} of V and some processor P_{r_T,c_T} of T, $r_V = r_T$ and $c_V > c_T$. Where V dominates T, we say that V* directly dominates *T if there is no task U such that V dominates U and U dominates T. The* direct dominance graph *is the directed graph having the collection of executing tasks as vertex set; for each pair of tasks T and V it contains an edge from T to V iff V directly dominates T.*

We build the direct dominance graph in $O(n^2)$ time in the following way. The list of executing tasks is sorted into increasing base column order, where if two or more tasks share a column, they are sorted into increasing row order. For each task we create a graph vertex and insert it in sorted order. A vertex already in the graph has associated with it the bottom– and topmost rows covered by tasks in its subgraph. Vertex insertion can therefore be done in linear time by a depth first search of vertices not visited before to determine whether the task is to the right of the subgraph or not. For each edge inserted, we associate the distance from the parent to the newly added child. After the graph has been built, we compute and store at each vertex the maximum distance the task can be moved to the right by summing the edge distances in a bottom–up fashion. Note that the distance the terminal nodes can be moved is given by their base columns and their widths. This final step, which takes $O(n)$ time, eliminates the need to search the subgraph of a vertex in order to determine how far it can be compacted each time we check the feasibility of a site. Figure 8 depicts the direct dominance graph for the arrangement of tasks in our example.

The graph can be searched in $O(n)$ time with each potential base location $b \in \mathcal{B}$ to determine whether the allocation site based at b can be freed of executing tasks by compaction. This procedure also involves a depth first search of those subgraphs whose covered rows intersect the allocation site based at b. Once the depth of the leftmost task(s) that intersects the allocation site is reached, each can be checked to determine whether it can be moved out

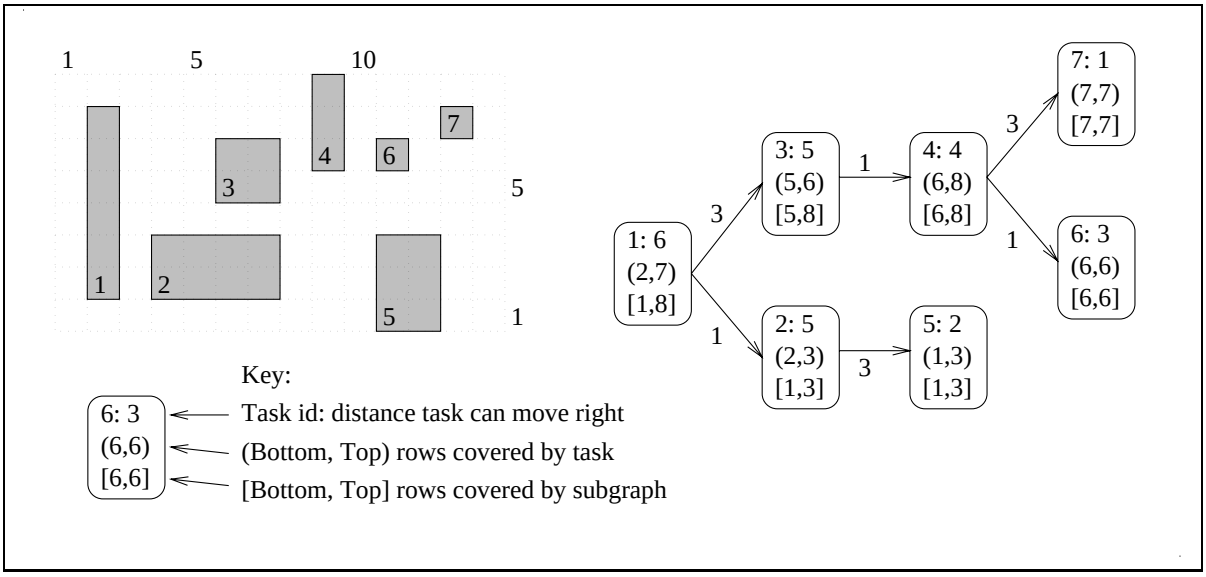

Figure 8: Arrangement of labeled tasks with its direct dominance graph.

of the way of the incoming task. By implication, if a task can be moved far enough to the right, its subgraph can as well.

With the means of identifying feasible allocation sites, we go on to describe three methods for scheduling the compaction.

3.2.5 Compacting Using Mesh Links

Given a set of tasks to be compacted, and for each, a distance it has to move, which is calculated by traversing the direct dominance graph, compaction by mesh links proceeds in the following way. The tasks to be compacted are simultaneously switched out of context and processors send their task elements to their mesh neighbours to the right. Processors receiving a task element check whether the task element has reached its destination and pass it onto the right if not. All task elements for a task arrive at their destination in the same cycle, after which the task elements are switched back into context to resume execution. Individual tasks are delayed for the time they are in motion, which is proportional to the distance they move. The time needed to free the allocation site is proportional to the distance the leftmost column of the leftmost occupying task has to travel.

3.2.6 Sequential Reconfigurable Compaction

This method attempts to delay the executing tasks as little as possible by three means: tasks are not switched out of context until moved; reconfigurable buses are used to move individual tasks in minimum time according to Lemma 1; and tasks are not moved onto executing tasks, where they would contend for use of the buses and use of the processors. These means are enforced by moving tasks according to the direct dominance graph: a task is not moved until all tasks in its subgraph have moved. Siblings may of course move in parallel. Individual tasks are delayed for the minimum time needed to move them, which is proportional to the distance they have to move, or their width, whichever is least. The

time needed to free the allocation site is proportional to the time needed to sequentially move the tasks on the critical path to the roots of the subgraph within the allocation site.

3.2.7 Parallel Reconfigurable Compaction

The allocation site is freed in minimum time by moving the leftmost task elements remaining within it in consecutive cycles. We apply this approach to the set of tasks that are to be compacted, whether they lie within the allocation site, or not. Consider a row of the mesh that intersects tasks that are to be moved. In each step we move the leftmost task element remaining to be moved. For a task element that moves in some cycle, we also move the task element at its destination, if it exists and has not moved yet, or the next task element to the right that is yet to move. In this way, the method makes greedy use of the reconfigurable row bus, and the compaction schedule requires minimum time to complete. However, some tasks may be delayed for the entire length of the schedule, which may exceed the minimum number of cycles required to move the task according to Lemma 1. The time needed to free the allocation site is proportional to the maximum number of task elements on any row that lie within it.

3.2.8 Selecting the Best Allocation Site

Given the means for assessing the cost of compacting a set of tasks using the compaction methods discussed in Sections 3.2.5 to 3.2.7, the feasible allocation sites can be compared on the basis of cost to free the site of occupying tasks in minimum time. For example, the compaction illustrated in Figure 5 is of minimal cost for the three methods discussed. The time required to free the allocation site using mesh links is 3 cycles. Using sequential reconfigurable compaction, 4 cycles are needed, while parallel reconfigurable compaction clears the allocation site in 2 cycles.

We have discussed partial ordered compaction to the right in this section. It is not difficult to establish that the minimum cost may not be as low as is possible for some compaction to the left, or to the top or bottom of the mesh. All four directions need to be considered if the compaction cost is to be minimized. However, the computational effort may in practice not be worth it.

4 Experimental Results

We conducted a series of experiments to evaluate the benefits of one–dimensional ordered partial task compaction over a range of process transfer costs. Requests for service consisting of task sizes and service times were derived from trace data obtained from a 400 node Intel Paragon at San Diego Supercomputer Center (SDSC) [1]. This data set was reduced from 33,343 records to 33,058 after removing records with unknown start times and records with processing times of less than one unit (second). The linearly given task sizes were converted into requests for the most square rectangles of the given size, if the rectangles did not exceed the dimensions of the mesh, or fitted into the most square rectangles that could accommodate the request when not. The observed performance in simulation experiments using this data set compares reasonably well with entirely random data while in addition removing the "unreality" of randomly generated task sizes and run times. The requests

were presented to a reconfigurable mesh simulator in their original order using uniformly distributed random intertask arrival periods. We examined the performance of the simulated system as the maximum intertask arrival period and the process transfer time per link was varied using three allocation and four compaction costing methods:

1. Bottom–Left (BL) — task were allocated to the bottom–leftmost free block of contiguous processors large enough to satisfy the request.

2. Non–contiguous (NC) — tasks were allocated non–contiguously whenever sufficient processors were available.

3. Bottom–Left with one–dimensional ordered partial task compaction — tasks were allocated BL whenever it was possible to do so. When not, compaction was performed, if possible, using the following cost methods:

   (a) Cost Free Compaction (CFC) — executing tasks were compacted *instantaneously*. Compaction costs were therefore not accounted for.

   (b) Mesh Link Compaction (MLC) — executing tasks were compacted using mesh links to transfer task element contexts, as described in Section 3.2.5.

   (c) Sequential Reconfigurable Compaction (SRC) — tasks used reconfigurable bus links to move according to the directed dominance graph, as described in Section 3.2.6.

   (d) Parallel Reconfigurable Compaction (PRC) — task elements were transfered greedily using reconfigurable bus links, as described in Section 3.2.7.

Although Section 3.2.8 describes how the best allocation site could be found for each compaction method, in this evaluation we allocated at the first feasible site found so as to reduce the simulation times. Nevertheless, with the aim of allocating blocked tasks as soon as possible, the simulator checked the feasibility of all possible allocation sites in all four directions with both possible task orientations whenever a task was deallocated.

The costs of finding allocation and compaction sites were considered negligible compared with the cost of loading and unloading a task or performing a compaction. They were therefore ignored. Since the cost of loading and unloading tasks is independent of the allocation method, these too were not included in the simulation. Our results reflect the cost of performing compaction for link transfer costs ranging from 10 μs to 10 ks per task element transfer. The unit of computation time was a second. We modeled the transfer time of a task element across a reconfigurable bus as a constant independent of distance, and equal to the delay incurred in traveling a mesh link.

In order to compare the performance of the allocation methods, a request's arrival time, the time its allocation commenced (request reached the head of the pending queue), the time its processing commenced (the request was satisfied), and the time its processing completed were logged. The results of 10 runs on a 20×20 reconfigurable mesh were averaged to obtain the results presented here.

4.1 Comparison of Allocation Methods

We first compare the performance of BL with CFC in order to gain an appreciation for the maximum possible performance benefit due to one–dimensional ordered partial compaction. The results for NC provide an upper bound on the benefit obtainable with any compaction method assuming the available processors could always be compacted. Note that the results for NC do not provide a tight upper bound, since the message contention and latency resulting from non–contiguous allocation increases the response time of affected tasks.

The *mean allocation delay* is the amount of time the request at the head of the pending queue waits for sufficient processors (for the allocation method) to become available to commence processing on average. The mean allocation delay for BL, CFC and NC is plotted in Figure 9. At maximum intertask arrival periods of less than 300s the system was saturated with work as tasks arrived more frequently than they could be allocated. The rate at which tasks were allocated was dependent on the ability of the allocation method to find a suitable free block for the task at the head of the pending queue. Only a fraction of blocks freed as a result of tasks leaving the system satisfied the next request. We therefore observed a constant allocation delay in the saturated system. While the next request was frequently blocked for BL, it was sometimes possible to combine the free processors to satisfy the next request with CFC, and with NC this was always possible once sufficient free processors became available.

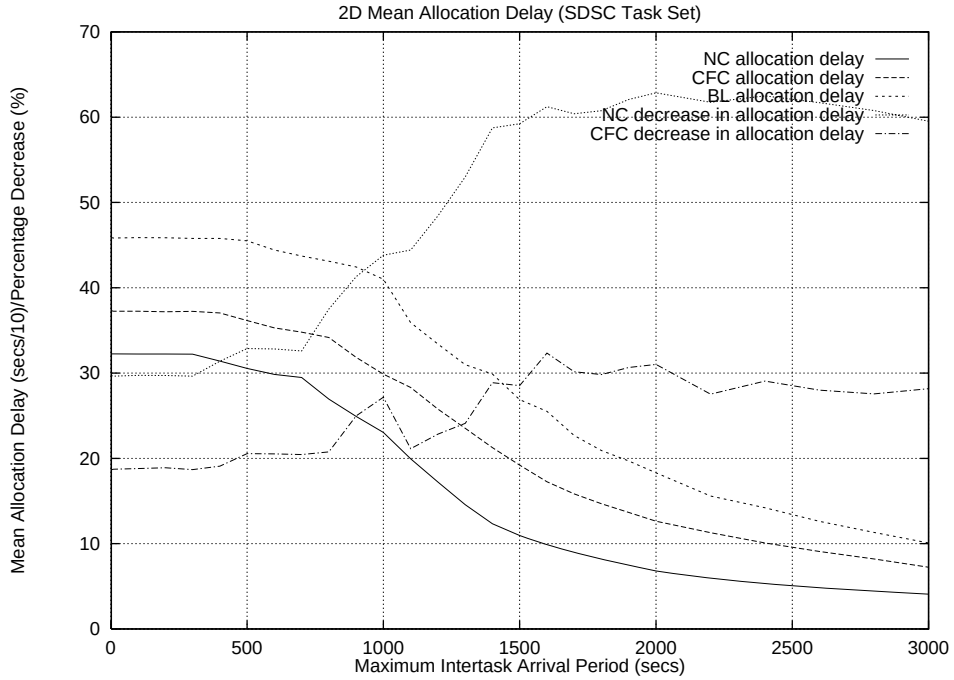

Figure 9: Mean allocation delay to SDSC tasks arriving at uniformly random intervals on a 20×20 mesh using Bottom–Left, Cost Free Compaction, and Non–Contiguous allocation methods.

The reduction in mean allocation delay at saturation due to CFC was approximately 18%, while with NC, the reduction was over 29%. These reductions increased as the system came

out of saturation due to an increased ability to overcome fragmentation problems when the packing of the mesh became less.

Positive feedback as a result of allocating tasks more effectively led to a reduction in the maximum intertask arrival period at which CFC and NC came out of saturation as indicated in the leftmost dips in their curves. NC, which has the least trouble allocating tasks, began coming out of saturation at a maximum intertask arrival period of approximately $300s$. CFC came out of saturation at a maximum intertask arrival period of about $400s$, while BL did not leave the saturated region until the maximum intertask arrival periods approached $500s$. These turning points correspond to the mean allocation delay at saturation.

When the system is saturated, the average amount of time a request spends advancing to the head of the pending queue, the *mean queue delay*, is proportional to the difference in the rate at which tasks arrive, and the rate at which they are allocated. This behaviour is observed in Figure 10, which plots the mean queue delays for BL, CFC and NC. In the saturated region, the benefit due to CFC rises from 20% for an intertask arrival period of $1s$, while for NC it increases from over 30%. As the system came out of saturation, the decrease in mean allocation delay and the increase in mean intertask arrival period contributed to reduce the mean queue delay. At high maximum intertask arrival periods, tasks were rarely queued because the allocation delay was small, even when successive tasks arrived shortly after one another

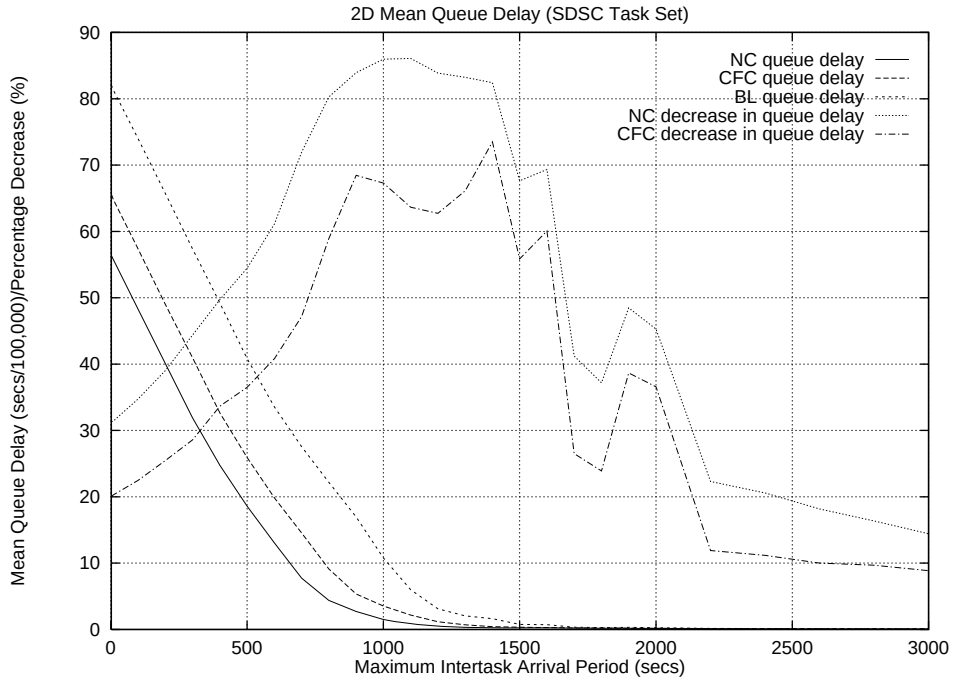

Figure 10: Mean queue delay to SDSC tasks arriving at uniformly random intervals on a 20×20 mesh using Bottom–Left, Cost Free Compaction, and Non–Contiguous allocation methods.

Figure 11 charts the *mean completion time* for the task set. The mean completion times were roughly constant in saturation and increased linearly with maximum intertask arrival period out of saturation. This is because during saturation the completion times depend

on the rate at which tasks can be allocated, while at higher intertask arrival periods, they depend on the rate at which tasks arrive. The benefit due to CFC was just over 18% in the saturated region, and that for NC was over 29%. As the system came out of saturation, the reduction rapidly fell to 0.

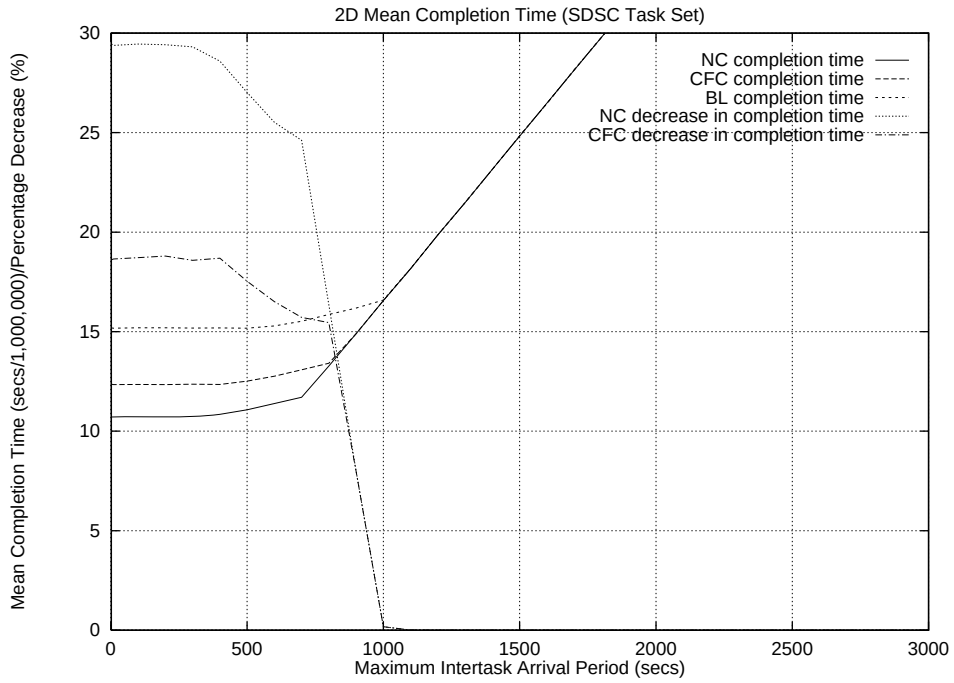

Figure 11: Mean completion time for SDSC task set arriving at uniformly random intervals on a 20×20 mesh using Bottom–Left, Cost Free Compaction, and Non–Contiguous allocation methods.

From Figure 12, we observe that the *utilization* of the array was constant for all allocation methods when the system was saturated. When the system was no longer saturated, the utilization became inversely proportional to the maximum intertask arrival period, since for a given amount of work, computing resource and allocation method, the utilization is inversely proportional to the completion time. In saturation, the utilization was therefore inversely proportional to the mean allocation delay, which was constant, and out of saturation, the utilization was inversely proportional to the maximum intertask arrival period.

The increases in utilization due to the CFC and NC allocation methods were approximately 23% and 42% respectively when the system was saturated. This benefit rapidly decreased to 0 as the system came out of saturation. Since CFC and NC reduced mean allocation delay but did not influence the intertask arrival period, compaction only improved utilization during saturation.

4.2 Comparison of Compaction Methods

In the previous section we found that the reduction in mean allocation delay at saturation is a good indicator for the reduction in mean queue delay and completion time at saturation. The percentage increase in utilization was found to exceed the percentage reduction in mean

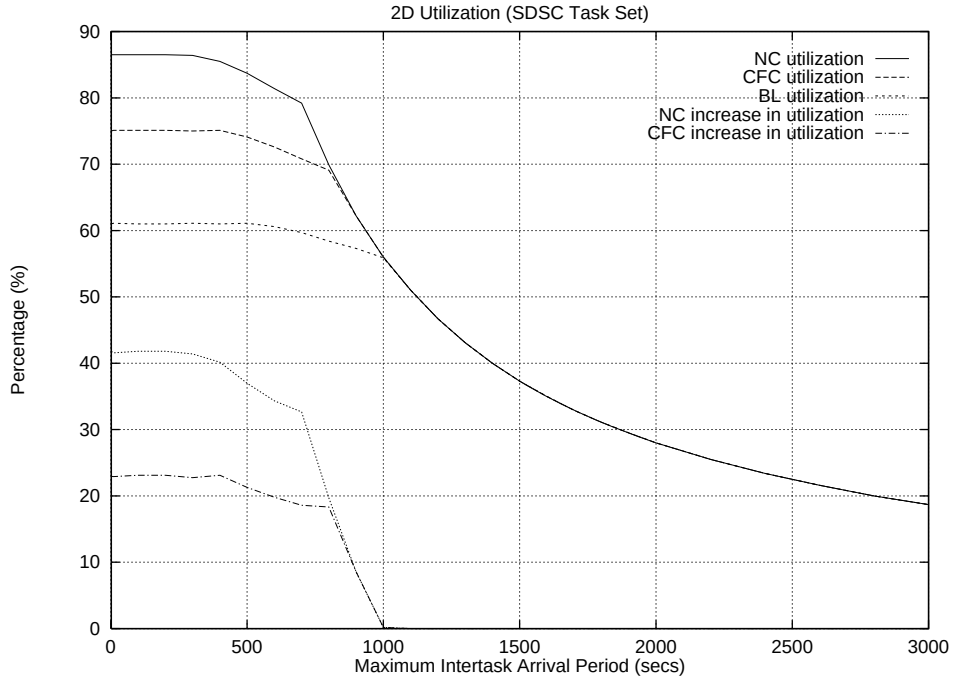

Figure 12: Processor utilization for SDSC tasks arriving at uniformly random intervals on a 20×20 mesh using Bottom–Left, Cost Free Compaction, and Non–Contiguous allocation methods.

allocation delay at saturation. In this section we compare the performance of the mesh, sequential, and parallel compaction methods for a range of transfer costs with that of Cost Free Compaction. The presentation is restricted to a comparison of mean allocation delays, which is indicative of the relative performance on other metrics.

We present plots of the mean allocation delay for link delays of 10 seconds/transfer, be it over a mesh link or reconfigurable bus, (Figure 13), 100 seconds/transfer (Figure 14), and 1,000 seconds/transfer (Figure 15). The mean allocation delay curve for the BL allocation method is included as a reference.

At a link delay of 10 seconds/transfer, we found that the performance of PRC, SRC and MLC was almost identical with that of CFC, which is free of transfer costs. Experiments at link delays of less than 10 seconds/transfer produced similar results. In this range, almost imperceptible variations in performance do occur when the delay to executing tasks results in differing task arrangements. When the cost was set to 100 secs/transfer we observed a 2% increase in mean allocation delay over CFC in all three methods. At a delay of 1,000 seconds/transfer, significant increases in the mean allocation delay were observed. However, PRC was still below the mean allocation delay of BL, despite the enormous link delay.

5 Discussion

The results for Cost Free Compaction indicate that significant performance gains can be obtained for one–dimensional ordered partial task compaction if the cost is low. We have found that compaction methods using link delays of less than $100s$, be it a connection to a

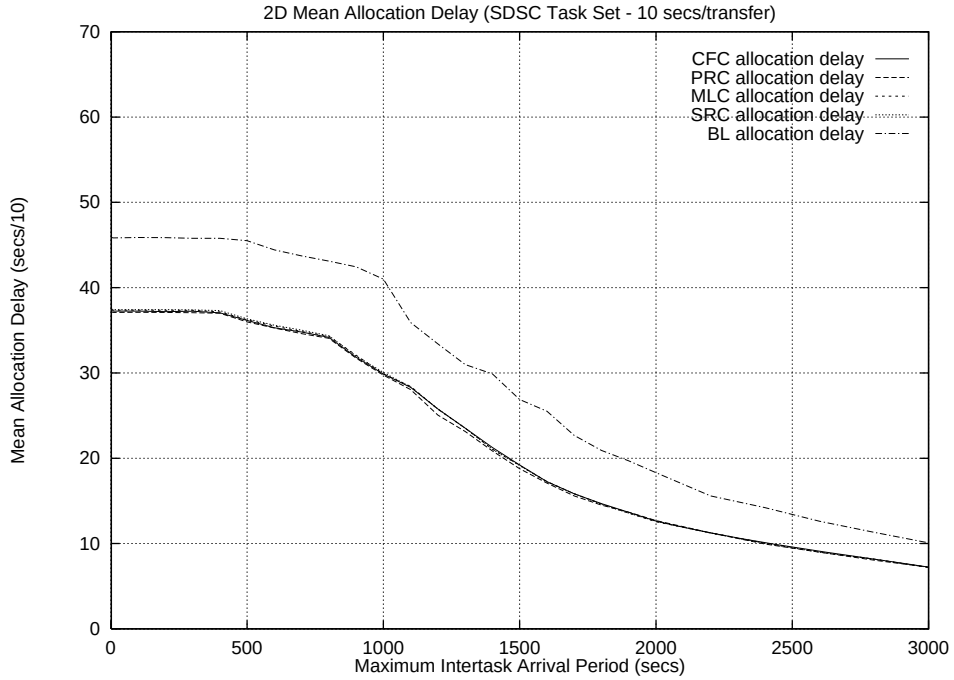

Figure 13: Mean allocation delay for SDSC tasks arriving at uniformly random intervals on a 20×20 mesh using Cost Free, Parallel Reconfigurable, Mesh Link, and Sequential Reconfigurable compaction methods at a cost of 10 seconds/transfer.

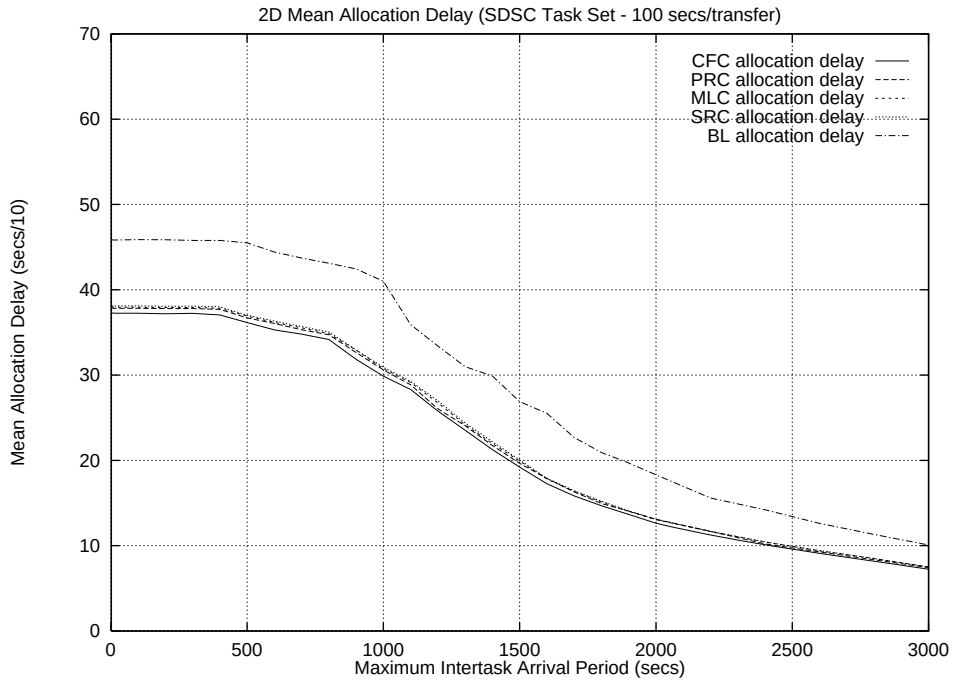

Figure 14: Mean allocation delay for SDSC tasks arriving at uniformly random intervals on a 20×20 mesh using Cost Free, Parallel Reconfigurable, Mesh Link, and Sequential Reconfigurable compaction methods at a cost of 100 seconds/transfer.

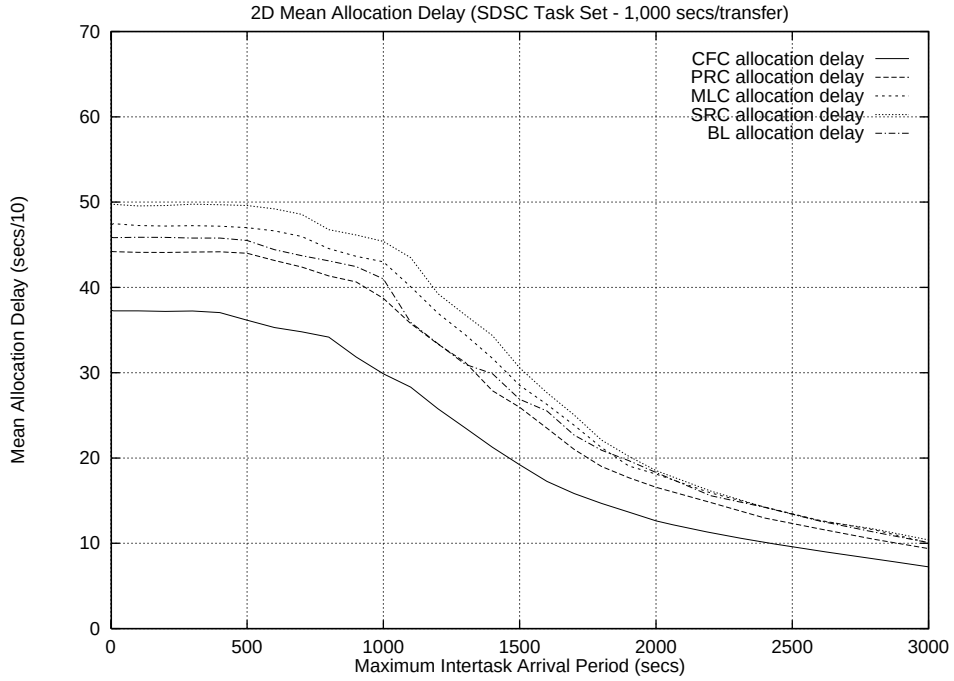

Figure 15: Mean allocation delay for SDSC tasks arriving at uniformly random intervals on a 20×20 mesh using Cost Free, Parallel Reconfigurable, Mesh Link, and Sequential Reconfigurable compaction methods at a cost of 1,000 seconds/transfer.

mesh neighbour, or a reconfigurable bus segment, yield performance benefits close to those of Cost Free Compaction. Surprisingly, this is the case even for partial compaction using mesh links, contrary to the popularly held view that task migration is costly in mesh–based parallel architectures. The factors influencing the maximum link delay sustainable without loss of performance remain to be identified, however, these may well include the distribution of requested service times. The performance gap between Cost Free one–dimensional ordered partial compaction and Non–Contiguous allocation indicates that there may be scope for improvement, perhaps through the use of multi–dimensional compactions or arbitrary rearrangements of tasks, depending upon how close the results for NC are to contiguous allocation with any compaction method.

The use of reconfigurable buses to transfer task elements was found to be only marginally better than using mesh links in the range of reasonable transfer costs, although some dependence on the distribution of task sizes may be possible. The benefit resulting from the use of buses should be greater when arbitrary rearrangements are considered, since it is more likely that tasks will be moved beyond their boundaries. A parallel compaction schedule that reduces the delay to executing tasks will probably improve the performance of Parallel Reconfigurable Compaction only slightly, since within the range of reasonable transfer costs, the Sequential Reconfigurable Compaction method that delays tasks as little as possible, performs almost as well as the parallel method that does not attempt to minimize the delay to executing tasks. In any case, the performance resulting from both of these methods is close to that obtained with Cost Free Compaction.

Further investigation is needed to determine whether greater benefits are possible through the selection of a compaction site that minimizes the cost of compaction rather than com-

pacting at the first site identified. Some tasks, especially long–lived ones, participate in many compactions and are therefore delayed on several occasions. Methods that avoid this problem are sought.

In [4] we found that performance gains from partial task compaction are dependent on the task size distribution. The benefits are not nearly as great when the task set contains a significant number of tasks requesting more than half of the total resource. Performance gains of approximately 20% were obtained on linear array systems using partial task compaction when the size of the requests was kept below one third of the total resource. In this study we have assessed performance with a task set, 0.2% of whose requests are for more than half of the total resource. To be confident of the expected gains from partial task compaction, the results of running the experiment on several field–collected data sets are needed.

6 Concluding Remarks

We found that partial task compaction reduces the allocation delay to tasks, which in turn reduces the queue delay and the time to complete a given task set, and increases the utilization of the compute resource. These benefits are greatest when tasks arrive more frequently than they can be processed although significantly reduced allocation and queue delays can be expected even when tasks arrive infrequently. It was found that allocating tasks with compaction increases the load that can be sustained by the system before it saturates. The load–bearing capacity was found to increase in proportion to the reduction in mean allocation delay. Performance benefits of over 18% were obtained at saturation for data derived from trace files in spite of link delays of up to $100s$. For transfer costs in this range, performance differences between simple compaction methods using mesh links and reconfigurable bus links were small. Since for large transfer costs the benefit is considerable, we feel that partial task compaction ought to be considered on MIMD meshes as a practical means of reducing fragmentation, for combining subtasks to reduce message contention and latency, and for unifying time–slots in gang–scheduled systems.

Many research problems remain to be solved. These include: examining the possibility of improving the time complexity of algorithms used to find good allocation sites; determining the scheduling complexity of one–dimensional ordered compaction on meshes, and developing better compaction scheduling algorithms; developing methods to relocate the tasks occupying an allocation site to arbitrary locations on the mesh; taking into account tasks with deadlines; and minimizing the delay incurred by individual tasks in the interests of group performance benefits.

Acknowledgments

This research was partially supported by grants from the Australian Research Council, and the RMC at The University of Newcastle.

References

[1] S. D. S. Center. Intel Paragon trace data. Available by ftp from `ftp.cs.uoregon.edu/pub/lo/trace/sdsc.tar.gz`.

[2] H.-L. Chen and N.-F. Tzeng. Task migration in hypercubes using all disjoint paths. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Sy stems*, pages 358–363, Oct. 1994.

[3] Cray Research, Inc. *Cray T3D system architecture overview*, Sept. 1993.

[4] O. Diessel, H. ElGindy, and B. Beresford-Smith. Partial task compaction reduces queuing delays in partitionable–array machines. Technical report 96-06, Department of Computer Science and Software Engineering, The University of Newcastle, 1996. Available by anonymous ftp:
`ftp.cs.newcastle.edu.au/pub/techreports/tr96-06.ps.Z`.

[5] C.-H. Huang and J.-Y. Juang. A partial compaction scheme for processor allocation in hypercube multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Proces sing*, volume 1, pages 211–217, 1990.

[6] K. Li and K.-H. Cheng. A two–dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system. *Journal of Parallel and Distributed Computing*, 12(1):79–83, May 1991.

[7] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor. The MAGIC VLSI layout system. *IEEE Design and Test of Computers*, 2:19–30, Feb. 1985.

[8] T. Schwederski, H. J. Siegel, and T. L. Casavant. A model of task migration in partitionable parallel processing systems. In *Frontiers 88: 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 211–214, Oct. 1988.

[9] T. Schwederski, H. J. Siegel, and T. L. Casavant. Task migration transfers in multistage cube based parallel systems. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 1, pages 296–305, Aug. 1989.

[10] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller Jr., H. E. Smalley Jr., and S. D. Smith. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, C–30(12):934–947, Dec. 1981.

[11] A. P. Sprague. A parallel algorithm to construct a dominance graph on nonoverlapping rectangles. *International Journal of Parallel Programming*, 21(4):303–312, 1992.

[12] Supercomputer Systems Division, Intel Corporation, Beaverton, OR. *Paragon XP/S Product Overview*, 1991.

[13] H.-y. Youn, S.-M. Yoo, and B. Shirazi. Task relocation for two–dimensional meshes. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 230–235, Oct. 1994.

[14] Y. Zhu. Efficient processor allocation strategies for mesh–connected parallel computers. *Journal of Parallel and Distributed Computing*, 16(4):328–337, Dec. 1992.