1 Appendix: Benefit of locality driven placement

We conduct the following experiment to study the benefit of locality driven placement on performance. Normally, Rawcc assigns instructions to virtual tiles, then it assigns virtual tiles to physical tiles by placing in close proximity virtual tiles that communicate a lot with each other. In this experiment, we replace locality driven placement with a random one, and we measure its effects on both execution time and route count, which is a count on the number of routes performed on the static network. The change in execution time measures the end-to-end benefit of locality-driven placement. The change in route count measures the increase in network traffic due to the loss of locality.

Table 3 presents the results of this experiment on 64 tiles. To help understand the results, the table includes the following data for the 64-tile base case: the level of exploited ILP, defined to be the speedup of the application on 64 tiles over a single tile; and the average route count per cycle.

Benchmark	Slowdown	Change in Route Count	ILP	Routes /Cycle
Sha	51%	89%	2.4	2.2
Aes	12%	68%	3.8	1.3
Fpppp-kernel	10%	52%	7.6	5.2
Adpcm	22%	75%	0.9	1.3
Unstructured	11%	187%	1.7	0.2
Cholesky	5%	9%	9.1	2.1
Vpenta	0%	0%	81.4	0.7
Mxm	0%	-23%	11.9	1.8
Swim	0%	79%	24.3	2.7
Jacobi	17%	122%	62.1	9.0
Life	54%	151%	48.7	13.9

Table 3. Impact of locality on performance.

For about one third of the applications, their performance is not adversely effected by randomized placement. These applications all have high ILP and low route count per cycle. Interestingly, an application may be sensitive to the method of placement either because it has low ILP, or because it has high route count per cycle. Applications with low ILP (Sha, Aes, Fpppp-kernel, Adpcm, Unstructured) only profitably uses a subset of those tiles. Randomizing tile placement thus destroys much of the application locality and leads to significant slowdown and higher network utilization. On the other hand, having a high route count indicates that an application communication a lot. Unless each tile communicates with every other tile with the same frequency, the application would benefit from locality driven placement.

The results demonstrate that fast access to a few tiles is preferable to uniformly medium-speed access to all the tiles. Thus, a mesh SON is likely to have better performance than a crossbar SON.

2 Appendix: Operand Analyses

To better understand the flow of operands inside an SON, we analyse the operand traffic at receiver nodes.

Analysis of Operand Origins Figure 16a analyses the origin of operands used by the receiver nodes. *Fresh* operands are computed locally, making use of 0-cycle local bypassing or the local register file; *remote* operands arrive over the inter-tile SON; and *reload* operands originate from a previous spill to the local data cache. The figure presents these results for each benchmark from two to 64 tiles. An operand is counted only once per generating event (calculation by a functional unit, arrival via transport network, or reload from cache) regardless of how many times it is used. Although these numbers are generated using Raw's SSS SON, the numbers should be identical for an SDS SON with the same 5-tuple. Not surprisingly, remote operands are least frequent in dense matrix benchmarks, followed by sparse matrix benchmarks, followed by the irregular benchmarks.

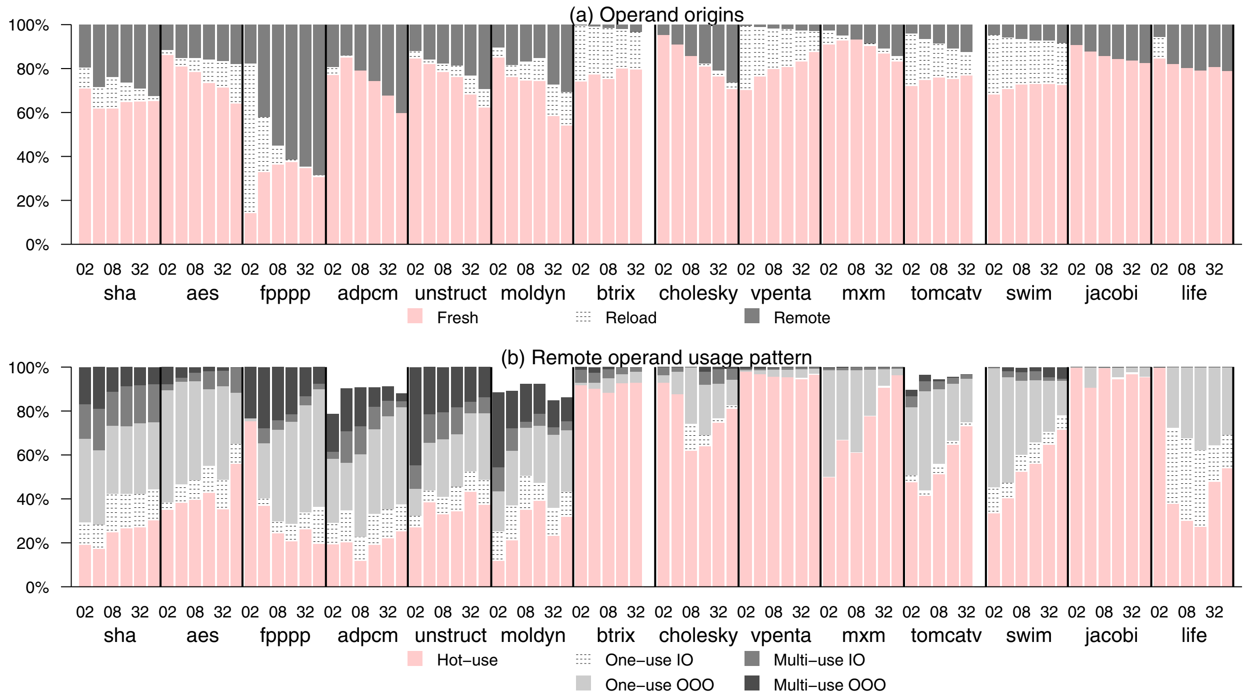

Figure 16. a) Analysis of operand origins. b) Analysis of remote operand usage pattern. For each benchmark there are six columns, one for each power of two configuration from two to 64.

Analysis of Remote Operand Usage Pattern We further analyze the remote operands from Figure 16a by their usage pattern. The data measures the ability of the compiler to schedule sender and receiver instruction sequences so as to allow values to be delivered just-in-time. Such scheduling impacts the optimal size of the IRF in the case of the SDS SON. In the case of the SSS SON, it reflects the importance of a facility to time-delay or re-cast operands, as mentioned in Section 6. This facility spares the compute processor from having to insert move instructions (in effect, a hidden receive occupancy) in order to preserve operands that arrive earlier than needed.

We classify the remote operands at the receiver node as follows. *Hot-uses* are directly consumed off the network and incur no occupancy. *One-uses* and *Multi-uses* are first moved into the register file and then used – *One-uses* are subsequently used only once, while *Multi-uses* are used multiple times. We also keep track of whether the operand arrives in the same order in which a tile uses them. An SON operand is *in-order (IO)* if all its uses precede the arrival of the next SON operand; otherwise it is *out-of-order (OOO)*.

Figure 16b shows the relative percentages of each operand class.[13] By far the most significant use-type are Hot-Use operands, followed by One-use OOO operands. We can compute the effective receive occupancy by summing the fraction of operands that are not Hot-Uses. This value varies greatly between the benchmarks, ranging from around 0 (100% Hot-uses) to 0.8 (20% Hot-uses). On 64 tiles, the average occupancy over all the benchmarks is 0.44. Recall that although this occupancy is less than one, Figure 10 shows that even a single cycle of occupancy can have a large effect on performance.

We intend to extend the Raw compiler so that it utilizes the static routers' existing register files and reduce this occupancy to nearly zero. These register files can be used to time-delay OOO operands, and to re-cast multi-use operands into the compute processor. Future work will determine the optimality of this structure.

[13]Due to how the compiler handles predicated code, a few of the benchmarks have SON operands that are not consumed at all, which explains why a few of the frequency bars sum up to less than 100%.