Self-Hosted Placement for Massively Parallel Processor Arrays

Graeme Smecher, Steve Wilton, Guy G. F. Lemieux
University of British Columbia
2332 Main Mall, Vancouver, BC Canada V6T 1Z4
gsmecher,stevew,lemieux@ece.ubc.ca

***Abstract*—We consider the placement problem as part of the CAD flow for a massively parallel processor arrays (MPPAs). In contrast to traditional placers, which operate on a workstation with one or several cores and are able to take advantage of parallelism to a limited degree, we investigate running the placer *on the target architecture itself*. As the number of processor elements (PEs) in such a device scale, so too does the computational power available to the placer. This natural scaling helps avoid the long runtimes that afflict FPGA flows.**

In this paper, we propose a distributed placer suitable to run on a MPPA. This placer takes advantage of local interconnect fabric, and may be efficiently coded on a simple, RISC-like core. We investigate the performance of this placer and compare it to traditional, simulated annealing-based placers using both unrealistic (but nearly optimal) and realistic (but suboptimal) annealing schedules.

On a simulated $32 \times 32 = 1024$-core MPPA, the proposed algorithm furnishes placements within 5% of the optimal placement quality – a level competetive with the realistic, traditional placer. To do so, the distributed placer requires each PE to consider 1/256th as many swaps as the traditional placer, a computational advantage which scales favourably as the number of cores on the MPPA increases.

I. Introduction

As silicon geometries shrink, power and design complexity become first-class design constraints. As a result, massively parallel processor arrays (MPPAs) have become a research focus as single-core processor performance ceases to scale aggressively. In contrast to single-core designs deriving their performance from advanced control logic (e.g. branch prediction, out-of-order execution, transactional memory, vector units, etc.), MPPAs feature hundreds or thousands of relatively simple cores arranged in a single-chip array.

In addition to academic and affiliated research projects (e.g. PiCoGA [1] and PACT XPP [2]), numerous commercial ventures (e.g. Nethra/Ambric, Tilera, Picochip, IntellaSys) have begun producing such devices. The number of cores they integrate is expected to scale rapidly.

Effective synthesis of programs for MPPAs remains a major challenge. Different approaches range from explicitly parallel programming models (such the approach used by Nethra/Ambric, in which numerous parallel programs are hand-coded in a Java-like language) and automatic synthesis and parallelization of code (e.g. [3], which synthesizes parallel programs from Verilog source code.) In most of these models, once the user program has been synthesized into a set of parallel processes, these processes must be placed and routed in a sequence analogous to traditional CAD flows for FPGAs. To greatly oversimplify, the major difference between MPPA and FPGA flows is the larger size and greater capabilities of each element in the architecture, and the correspondingly lower number of elements.

In this paper, we consider fast placement algorithms for MPPAs. Since the problem is similar to FPGA placement, any placement algorithm suitable for FPGA flows may readily be adapted to MPPAs. Moreover, since the number of cores in such an architecture is several orders of magnitude lower than the number of placement elements (i.e. clusters) in an FPGA, such an approach would furnish a CAD flow with greatly reduced execution times. In order to shorten placement time further, we investigate a distributed placement algorithm intended to run on the MPPA itself. We do so for several reasons:

- We prefer to look to compilers (instead of FPGA flows) as a barometer for reasonable synthesis times,
- As MPPA sizes scale upwards, placement complexity may again become a significant problem,
- Fast placement will motivate research into more intelligent tools (e.g. combined placement and routing), and
- A distributed approach in which the target silicon *performs its own placement* is interesting in its own right.

In the following section, we review previous research into distributed placement algorithms. Then, we introduce a distributed simulated annealing algorithm suitable for running on a MPPA.

II. Background

In this section, we review several approaches to distributed placement. We restrict our focus exclusively to approaches based on Simulated Annealing (SA), since it is an established placement technique with a significant history in the FPGA community. There are few published placement algorithms specifically targeting MPPA flows, and especially for our purposes, SA's strengths as a general technique that may easily be extended to arbitrary placement constraints (e.g. to manage any architectural quirks of a particular MPPA) is desirable. A familiarity with simulated annealing is assumed; for an overview, please consult the references.

Several early works considered distributed simulated annealing via shared-memory and message-passing multiprocessors.

In [4], placement cells are divided dynamically between processors. Each processor may only select and move cells it 'owns.' Runtime speedups of 3.3 (using 4 processors) and 6.4 (with 8 processors) were reported. In [5], annealing is parallelized differently during two distinct phases: in the high-temperature phase, where the probability of accepting an unfavourable move is substantial, each processor investigates a completey distinct placement. After these placement have been independently refined, the best is selected for the low-temperature phase, during which each processor is assigned a geometric partition of the chip. Each processor then anneals this partition, swapping nodes with the processors working on adjacent partitions. Speedups are up to 4.3 on a 5-processor system, and are projected to 7.1 on a 10-processor system. These early works and others (e.g. [6]) are characterized by the use of relatively few processors, and do not investigate scaling to hundreds or thousands of processors.

More recently, a distributed placement algorithm for FPGAs which runs on an FPGA accelerator has been proposed [7]. In this paper, placement occurs on a systolic architecture in which each placement node (i.e. a cluster) is assigned its own processor, and is permitted to communicate only with its four nearest neighbours. The topology and detailed architecture of this approach are shown respectively in Fig. 1 and 2. The placement architecture was described in an HDL and synthesized for FPGAs; as each processor element (PE) in this architecture requires many more FPGA resources than it manages, this approach required multiple FPGAs to generate placements for a single one.

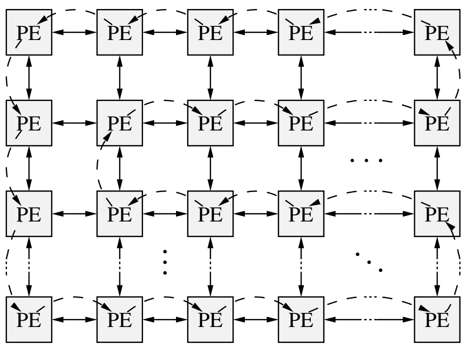


Fig. 1. Top-level architecture of distributed SA from [7]. Each square represents a processing element in the systolic array. Nearest-neighbour links are shown using solid arrows; the position update chain (which furnishes PEs with estimates of block locations) is dashed.

In the following section, we adopt a similar approach to placement on MPPAs. In doing so, we find that each core is suitably powerful to manage placement of a block its own size.

III. Proposed Algorithm

Figs. 3 and 4 show the high-level and detailed MPPA described in [3]. This architecture consists of an array of identical processor elements (PEs) and a local routing fabric. Each PE is a simple, 32-bit RISC-like core executing a program stored on local memory. Programs for each PE, as well as a static routing schedule for inter-PE communications, are synthesized

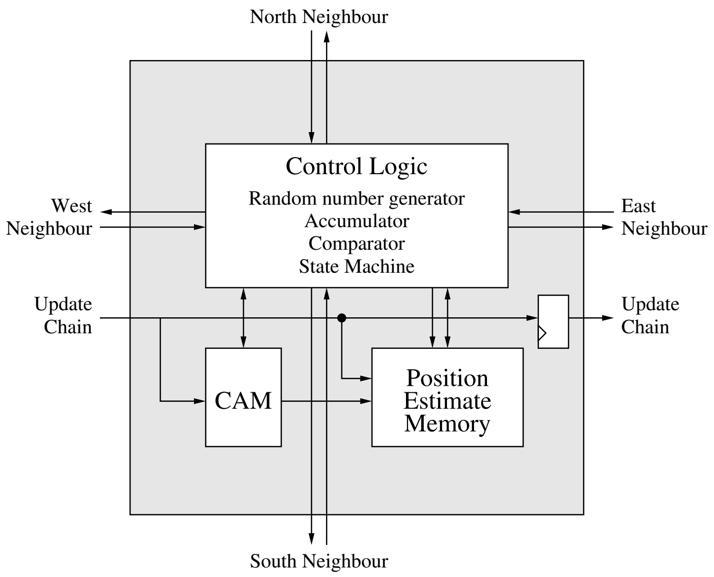


Fig. 2. Architecture of a single PE from [7]. Note the reliance on content-associative memory (CAMs) for position look-ups.

from Verilog RTL. The choice of this particular architecture is arbitrary, in a sense: Subject to per-node memory requirements (which are examined below), this investigation is equally relevant to other MPPA architectures.

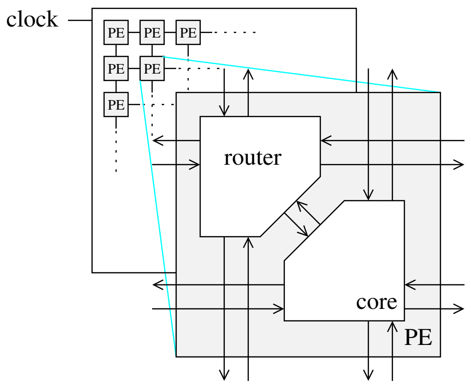


Fig. 3. Top-level architecture of MPPA in [3]

We propose a simulated annealing algorithm with the following characteristics:

- Each PE is assumed to be a traditional RISC-like core, with a moderate amount of local memory and without specialized structures (e.g. CAMs),
- The computational architecture (i.e. the resources used to compute placement) are structured identically to the placement problem. In other words, an array of size (x, y) is used to route a netlist involving at most $x \times y$ nodes.
- The SA implementation is conventional (i.e. it includes an exponential function to relate probability of swap acceptance, temperature, and difference in cost.)
- Communication between PEs is restricted to their immediate neighbourhood (which we will define below.)

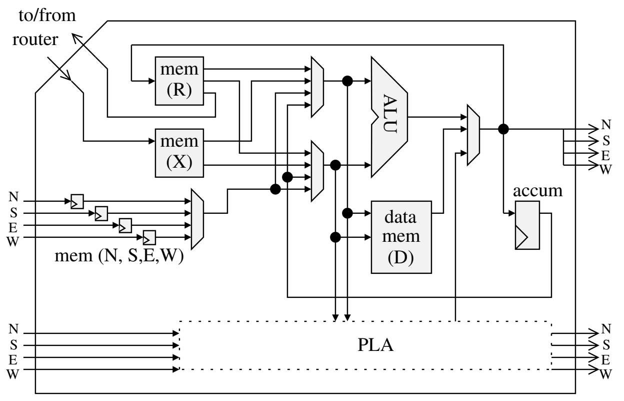


Fig. 4. Architecture of a single PE from [3]

In [7], the number of logic resources used in an FPGA to compute placement of a single cluster were much larger than the clusters themselves. In contrast, each PE in an MPPA is capable enough to handle placement of an element its own size. Thus, the placement problem scales exactly with the architecture. The high-level pseudocode is similar:

```
generate_random_placement()
**for** interval in 0 to TMAX **do**
  **for** each PE **do**
    **for** n=1 to 'updates' **do**
      Update position chain
    **end for**
    **for** n=1 to 'swaps' **do**
      Consider swaps with each PE in our neighbourhood
    **end for**
  **end for**
**end for**
```

This algorithm operates deterministically, and the placement results are reproducible. The second 'for' loop (which loops over each PE) is done in parallel, i.e. the loop's contents define the program which runs independently on each PE.

In the following subsections, we provide some details on each of the steps. First, we describe the data structures used to track annealing and qualitatively describe the memory requirements of the algorithm.

A. Data Structures

We use the four structures shown in Fig. 5 to track PE contents:

pbm The Placement-to-Block Map maps each PE's (x, y) location to the ID of the block it contains, or a token if the block is unoccupied.

bnm The Block-to-Net Map maps each block ID to the IDs of every net with which it connects.

nbm The Net-to-Block Map maps each net ID to the IDs of every block with which it connects.

bpm The Block-to-Placement Map maps each block ID to the (x, y) location of the PE it currently occupies.

These same structures are applicable to an ordinary annealing implementation. We use forward- and reverse structures in ordinary RAM to avoid either costly list searches or special structures such as associative memories that would not be available in a generic MPPA.

Of these structures, the net-to-block mapping (nbm) and the block-to-net mapping (bnm) encapsulate the block and net connections of the desired routing and are static throughout the annealing process. The other two (the block-to-placement map or bpm, and the placement-to-block map or pbm) link blocks with their (estimated) locations in the processor array, and are dynamic. Moreover, each PE's knowledge of blocks' locations is only approximate; their bpm and pbm structures are local and not synchronized with other PEs.

Assuming we allocate 16 bits for net and block IDs, and 16 bits for placement (8 bits each for the horizontal and vertical position), each single entry in the pbm and bpm occupies 32 bits. Such an allocation scales to 65536-core processors in a 256×256 array. Each entry in the nbm and bnm requires a variable number of 16-bit entries, depending on connectivity. It is unreasonable to store a complete version of *any* of these structures in each PE (for example, a complete pbm requires $256 \times 256 \times 2 = 131$ kB, which is a trivial amount for a PC but an order of magnitude larger than PE memories in a typical MPPA.) Fortunately, a given PE only accesses the entries in its own connectivity structures (the bnm and nbm) that are relevant to the block it contains; other entries are unaccessed and need not be stored. A strategy for avoiding full placement maps (the pbm and bpm structures) is described in [7]; this approach does impact the accuracy of each PE's knowledge of its neighbouring blocks, and requires transfers of moderately large data structures during block swaps. We assume complete copies of each structure are available to each PE, and thus model the effects of information staleness (but not the effects of limited per-PE memory.)

B. Position Chain Updates

Any distributed annealing algorithm requires some method to ensure all processors have some "image" of the placement state, and that each processor's image is updated in some fashion. It is not necessary that these images be either synchronized or correct, provided they are eventually updated and the impact of stale information is limited.

Here (as in [7]), each PE participates in an "update chain", a ring that passes through each PE once and snakes its way through the topology. At each iteration of the inner loop, each PE passes information along the chain, updating its internal structures with information about block placement. When information about a particular block arrives at the PE containing it, this PE absorbs the stale data and pushes out new information. Using this mechanism, position updates occur with only nearest-neighbour communications.

The pseudocode for updating each PE is as follows:

```
(x,y,bid) = local_uq.deq()
**if** (x,y) == (local_x, local_y) **then**
  bid = local_bid
**end if**
update local PBM, BPM
```

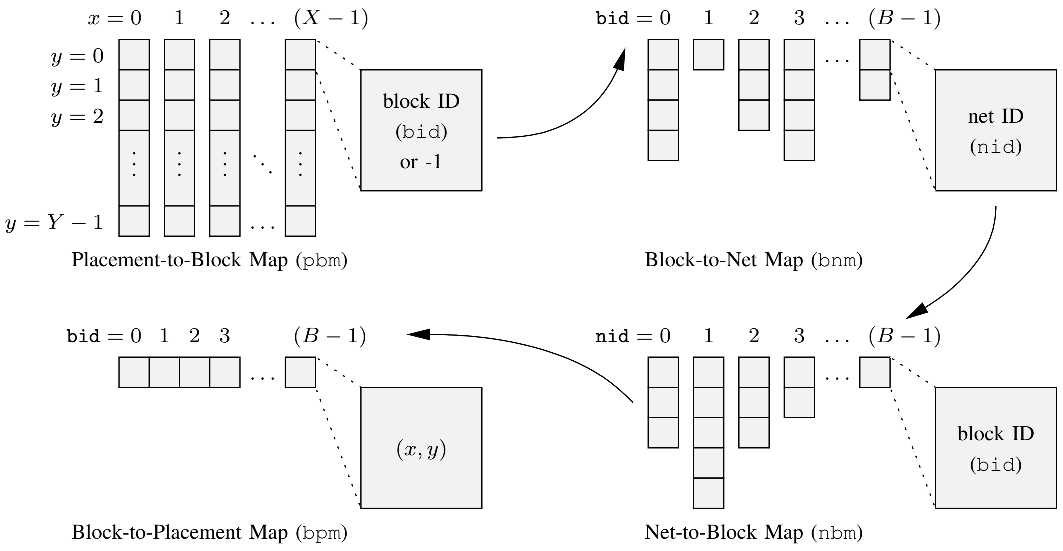


Fig. 5. Data structures used in distributed annealing algorithm

next_uq.enq(x,y,bid)

where 'local_uq' and 'next_uq' are input queues for the current and next PE in the update chain. The cost of maintaining the update chain is constant, and largely depends on the queueing or local routing hardware available.

In [7], the update chain takes a zigzag geometric loop through each PE. Here, because PE geometries change with each placement scenario, we have adopted a linear scheme that traverses each row, left-to-right, before jumping to the first column in the following row. This scheme is not necessarily appropriate for fabrication, but allows us to evaluate scenarios in which such a circuit cannot be constructed (i.e. with an odd number of rows or columns.) Because simulations do not suggest that stale placement information is a problem, we have not investigated other update-chain policies.

C. Neighbour-Neighbour Swaps

After the position chain has been updated, each PE considers block swaps with its neighbours. This process occurs in a synchronized fashion in a manner much like square dancing: every PE begins by pairing itself with a neighbouring PE and exchanging swap cost information. Paired PEs agree on a random number (using local, simple pseudo-random number generators) and use this number to agree on whether the swap is accepted. If so, the PEs exchange block IDs and any relevant data structures used to track them. PEs then change partners and start again.

In [7], swaps are considered for the 4 neighbouring PEs on the north, east, south, and west sides. As we will see, the use of these 5-PE neighbourhoods reduces placement quality; accordingly, we investigate two larger neighbourhoods: an 9-PE neighbourhood adds the possibility of swapping each block with PEs to the northeast, southeast, southwest, and northwest; and a thirteen-PE neighbourhood adds the possibility of swapping each block with PEs *two* blocks to the north, east, south, and west. These neighbourhoods are illustrated in Fig. 6. The use of larger neighbourhoods is particularly appropriate given the MPPA shown in Fig. 3, since the dedicated routing network may forward communications to nearby PEs without requiring cooperation from PEs along the route, and with only a small latency penalty.

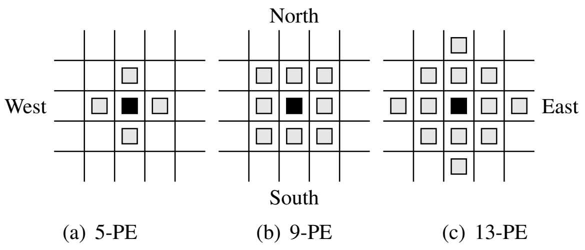


Fig. 6. Five-, Nine-, and Thirteen-PE Neighbourhoods. PEs correspond to grid locations; a given PE (black) may exchange blocks with each of its neighbours (grey) once during a single swap round.

Pseudocode for the swapping phases is as follows:

```
for each neighbour do
  $\Delta C$ = calculate_swapped_cost() - our_old_cost
  if random() $< e^{-\Delta C/T}$ then
    swap()
  end if
end for
```

For each swap, the change in global cost is the change in the half-perimeter bounding box for each net connected to the two blocks undergoing swapping. This requires each PE to track the approximate position of all blocks with which its block is connected. These positions need not be accurate, and each PE may use its own (possibly different) estimate without problems; as the temperature is reduced and fewer swaps are accepted, each PE's position estimates converge to the 'true' state. Although the pseudocode indicates floating-

TABLE I
BENCHMARKS (WITH BEST-CASE PLACEMENT COSTS)

Benchmark	**Blocks**	**Nets**	**Cost**
me	1024	998	1,242
dir	1024	760	1,785
chem	1024	749	1,250
mcm	256	244	404
honda	256	240	379
pr	256	128	181

point calculation, a fixed-point implementation could readily be obtained (e.g. by taking the logarithm of both sides, and using a log-weighted pseudorandom number generator.)

The computational cost of this stage depends on several factors, including the net-to-net connectivity (over which we iterate in calculate_swapped_cost) and the routing fabric. In our implementation, we assumed no limits on per-PE memory, which reduces the amount of neighbour-neighbour communication during a swap (only the local swap costs and block IDs need to be transferred.) If placement or connectivity information at each node is incomplete, these structures must also be swapped, and the neighbour-to-neighbour transfer speed (as well as the amount of PE supervision required) becomes important.

IV. RESULTS

In this section, we present some results from comparing a traditional (sequential) implementation of distributed annealing with our approach.

Both annealing algorithms were evaluated using six benchmarks. Five of these (`chem`, `dir`, `honda`, `mcm`, and `pr`) are DSP- or dataflow-style benchmarks taken from [8]. The final benchmark (`me`) is a motion-estimation algorithm, exhaustively searching for the offset of an 16×16-pixel subimage within a 32×32-pixel image with the minimal sum of absolute differences (SAD). All benchmarks are described in behavioural Verilog. From there, our CAD flow synthesizes Verilog into a dataflow graph (DFG) using a RISC-like instruction set. Nodes in the DFG are clustered, forming a program for each PE in the MPPA. The placement step requires each instruction cluster (block) to be assigned to a physical PE.

Benchmarks are listed, along with the half-perimeter cost of the best placement,[1] in Table I. Each placement has targeted a 32×32 architecture for a total of 1024 cores.

We evaluate the two placers by varying their parameters and investigating the corresponding change in placement quality. As the placement cost for each benchmark vary wildly, we have normalized each result to the placement cost in Table I; 1.0 indicates placement with excellent quality, and higher numbers indicate worse results. The default parameters used in the experimental results below are shown in Table II.

Since the "swaps" parameter (the number of node swaps considered at each temperature step) has a different meaning

[1]Generated via an extremely slow annealing schedule with $\alpha = 0.99$, $T_0 = 100$, $T_{\text{stop}} = 10^{-4}$, and $200,000$ swaps per temperature step.

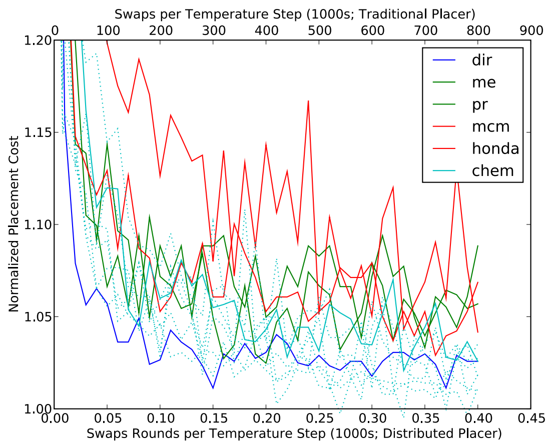
TABLE II
DEFAULT ANNEALING PARAMETERS

Param.	**Value**	**Description**
α	0.985	Rate of exponential decay in T
T_0	50	Starting temperature
T_{stop}	0.01	Ending temperature
Sequential Annealer		
swaps	2048×250	Swaps per temperature step
Distributed Annealer		
swaps	250	Swap rounds per temperature step
updates	20	Update-chain shifts per swap
neighbours	12	Blocks neighbouring each PE

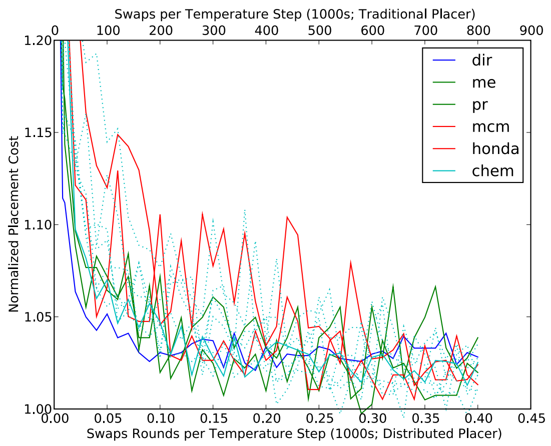
for the distributed and traditional placers, we have scaled these parameters to give a consistent *total* number of swaps for the entire architecture. For the traditional placer, $2048 \times 250 = 512,000$ swaps are considered at each temperature step. The distributed placer considers only 250 "swap rounds" per core at each temperature step; since each swap round includes a possible swap with each block in its neighbourhood, the overall number of swaps for the distributed placer is $250 \times 12 \div 2 = 1500$ (where 12 is the number of neighbours, and the division by 2 reflects pairing of nodes – it requires two nodes to consider a single swap.) This scaling permits results for both placers to be shown on the same graph.

To convert these figures into concrete units (i.e. placer runtime in seconds), more information about each placer's architecture (clock rate, interconnect bandwidth, local memory) are required. However, it is already clear that the computational requirements of distributed placement are several orders of magnitude lower than the traditional algorithm due to the reduced per-core swaps. (The fast clock and high IPC of modern workstation CPUs, relative to commercially available MPPAs, makes up some of this gap.)

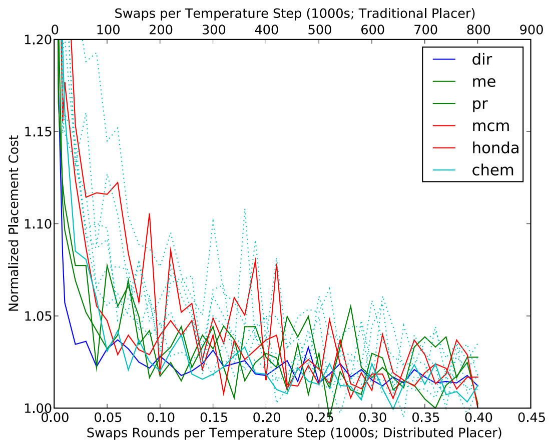
We begin by investigating the relationship between placement quality and the number of swaps. With the 5-PE neighbourhoods shown in Fig. 6(a), the placement quality in Fig. 7(a) is frequently over 10% worse than the reference solution and over 5% worse than the traditional placer. It is possible that with only four nearest neighbours to choose from, blocks become "fenced off" or trapped when their neighbours have highly favourable placements. In other words, there is insufficient *path diversity* for a given block to migrate to a favourable location in the presence of obstacles. A traditional placer, in which blocks are selected randomly and may swap independently of their relative proximity, does not suffer from this syndrome. Circumventing blockages can be difficult, unlikely, or effectively impossible for blocks depending on how content their neighbours are with their current location.



(a) 5-PE neighbourhoods



(b) 9-PE neighbourhoods



(c) 13-PE neighbourhoods

Fig. 7. Placement cost; Swaps parameter is varied. Distributed results use solid lines; traditional results are dotted.

With the 9-PE neighbourhood shown in Fig. 6(b), placement quality varies with the number of swaps as shown in Fig. 7(b). Path diversity (and thus placement quality) improves for most benchmarks. When a large number of swaps per temperature step are permitted, the distributed and traditional placer perform equivalently.

With the 13-PE neighbourhoods shown in Fig. 6(c), placement quality is with 5% of the optimal placement and competetive with the traditional placer. These results, shown in Fig. 7(c), suggest that adequate path diversity exists and that a favourably placed block no longer impedes movement of nearby blocks.

In terms of computational requirements, the distributed placer using 5-PE neighbourhoods considers a total of $250 \times 4 \div 2 = 500$ swaps per temperature step. The 9- and 13-PE neighbourhoods, respectively, require 1000 and 1500 swaps per temperature step, a doubling and tripling of swap effort. Compared to the traditional placer, which performs comparably, each PE in the distributed placer only considers a tiny fraction of the total number of swaps: respectively, $1/1024^{\text{th}}$, $1/512^{\text{th}}$, and $1/256^{\text{th}}$ as many. This advantage of the distributed placer over the traditional one improves as the size of the MPPA increases.

Fig. 8 shows the normalized placement costs as the α parameter (which controls the rate of temperature decrease) is varied from 0.95 to 0.999. Dotted lines represent the traditional placer, and solid lines represent the distributed algorithm. Both placers exhibit better-quality results with higher α, as is expected. Note that with very high α, experimental results actually dip beneath the 1.00 threshold, indicating that our estimated 'best-case' placements were not actually optimal.

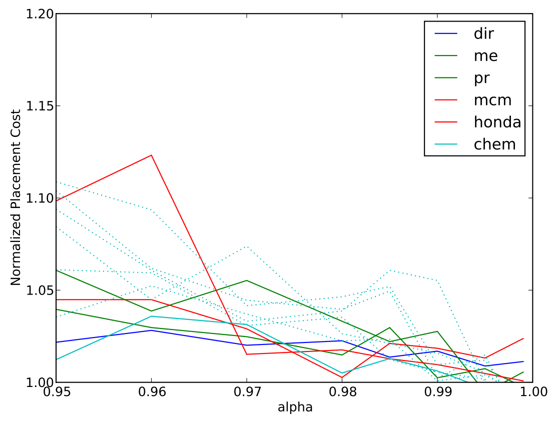


Fig. 8. Placement cost; α parameter is varied. Distributed results use solid lines; traditional results are dotted.

Fig. 9 shows the normalized placement cost as the starting temperature (T_0) is increased from 5 to 50. With a sufficiently high T_0, it is difficult to pick a trend out of this graph (except that both placers are extremely sensitive to starting conditions.)

Fig. 10 shows the implementations' sensitivities to the stopping temperature T_{stop}. The distributed placer is able to produce excellent placement results with a slightly higher T_{stop} than the traditional one, further increasing its computational advantage. The cause is straightforward: when T_{stop} is sufficiently low, non-greedy swaps effectively cease to

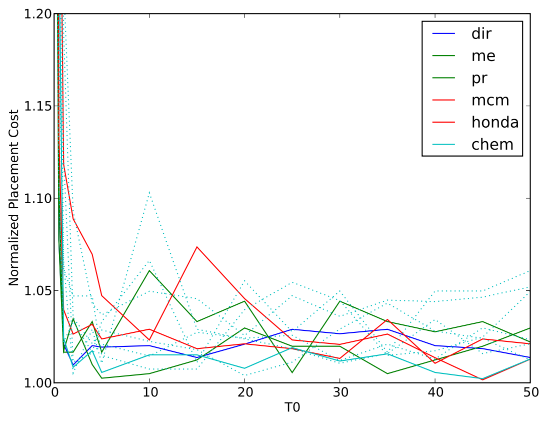


Fig. 9. Placement cost; T_0 parameter is varied. Distributed results use solid lines; traditional results are dotted.

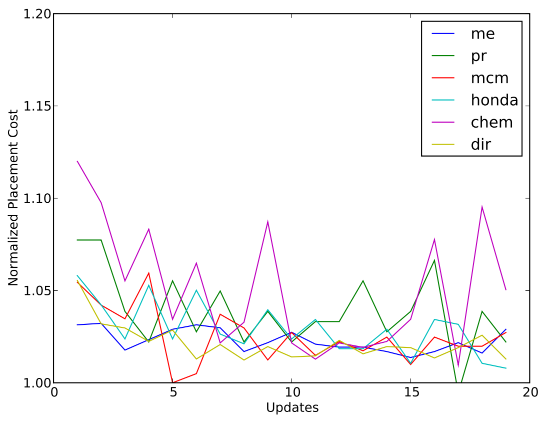


Fig. 11. Placement cost; 'updates' parameter is varied. Only the distributed placer is shown; the traditional placer has no such parameter.

occur in both placers. However, in the traditional placer, swaps are considered randomly and there is no guarantee that every advantageous swap is actually evaluated. Adding several temperature steps with negligible probability of accepting non-greedy swaps permits more greedy swaps to be considered. These extra temperature steps are not required by the distributed placer.

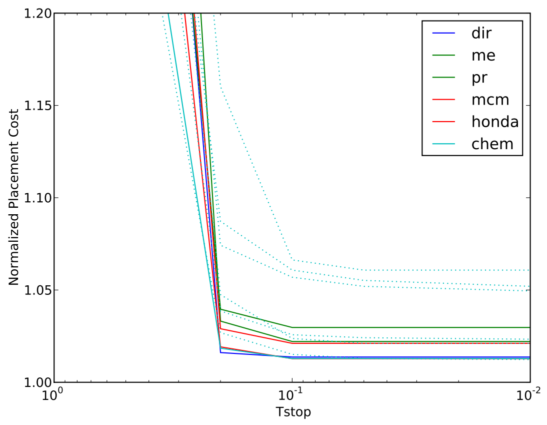


Fig. 10. Placement cost; T_{stop} parameter is varied. Distributed results use solid lines; traditional results are dotted.

Fig. 11 shows the normalized placement cost as the 'updates' parameter is varied from 1 to 19. This parameter determines the quality of each block's estimates of its neighbours' positions. Unless otherwise stated, the simulation results shown use 13-block neighbourhoods and 20 updates per swap round, suggesting just over 1 position-chain update per swap. Fig. 11 suggests this is an adequate balance between information staleness and extra data-structure maintenance.

V. Conclusions

In this paper, we explored the potential for MPPAs to perform self-hosted placement, i.e. for such a device to be used as part of its own CAD flow. We showed how a distributed annealing algorithm can furnish placements with comparable quality to ordinary simulated annealing, with a number of iterations that is several orders of magnitude lower, and with an architecture that scales naturally with the number of elements being placed.

Overall, the distributed algorithm was able to provide competetive results with a vastly reduced workload per PE. Performance was within 5% of the best-case placements obtained, and competetive with a traditional placer using a practical schedule, but requiring orders of magnitude less calculation per core. (At the operating point shown in Table II, the per-PE placement program required 1/256th as many swaps per temperature step, compared to the traditional placer.) Not only were the per-core swaps reduced, but the distributed placer's stopping temperature could be raised significantly without negatively impacting performance. In contrast to earlier work on distributed placement [4], [5], [6], the extremely high processor count, as well as the low latency and high interconnect bandwidth of MPPAs provides encouraging results and excellent parallelism.

We expanded the notion of a PE's 'neighbourhood' (the nearby PEs with which it may swap blocks directly) beyond the four PEs immediately to its north, south, east, and west. We found that larger neighbourhoods were a key to avoiding "stuck blocks" whose placements were suboptimal, but whose movement was impeded by immediate neighbours with satisfactory placements.

We considered the computational cost of the algorithm in generic terms, by exploring the number of swap rounds required for a given level of performance. A natural extension of this work involves mapping it to a specific architecture, allowing a characterization of the placer's performance in concrete units. Such an investigation requires some additional work on memory-efficient packing and fast exchanges of each PE's structures.

Several other avenues of investigation include better cost functions (e.g. to improve power consumption or routability).

VI. Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] A. Lodi, L. Ciccarelli, C. Mucci, R. Giansante, and A. Cappelli, "An embedded reconfigurable datapath for SoC," in *Proc. 13th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2005, pp. 303–304.

[2] M. Ganesan, S. Singh, F. May, and J. Becker, "H. 264 decoder at HD resolution on a coarse grain dynamically reconfigurable architecture," in *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2007, pp. 467–471.

[3] D. Grant and G. Lemieux, "A spatial computing architecture for implementing computational circuits," in *CMC Microsystems Annual Symposium*, 2008.

[4] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A parallel simulated annealing algorithm for the placement of macro-cells," vol. 6, no. 5, pp. 838–847, September 1987.

[5] J. Rose, W. Snelgrove, and Z. Vranesic, "Parallel standard cell placement algorithms with quality equivalent to simulated annealing," vol. 7, no. 3, pp. 387–396, March 1988.

[6] P. Banerjee, M. Jones, and J. Sargent, "Parallel simulated annealing algorithms for cell placement on hypercube multiprocessors," vol. 1, no. 1, pp. 91–106, Jan. 1990.

[7] M. G. Wrighton and A. M. DeHon, "Hardware-assisted simulated annealing with application for fast FPGA placement," in *Proc. ACM/SIGDA Int. Symp. on FPGAs*, 2003.

[8] M. Srivastava and M. Potkonjak, "Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput," vol. 3, no. 1, pp. 2–19, 1995.