

# Towards Scalable Placement for FPGAs

Huimin Bian, Andrew C. Ling, Alexander Choong, and Jianwen Zhu  
Department of Electrical and Computer Engineering  
University of Toronto  
Toronto, ON, Canada, M5S 3G4  
{hbian|aling|achoong|jzhu}@eecg.toronto.edu

## ABSTRACT

Placement based on simulated annealing is in dominant use in the FPGA community due to its superior quality of result (QoR). However, given the progression of FPGA device capacity to the order of 100K LUTs, the long runtime associated with simulated annealing warrants a revisit of other placement paradigms in the context of FPGAs. In this paper, we attempt to make a rigorous comparison of a recent crop of academic ASIC placers and VPR when applied to modern FPGA device features and design sizes. We also report a new detailed placer, MDP, based on a new problem formulation of maximum-bipartite matching. We show that MDP is 3X to 7X faster than the detailed placer in FastPlace, which until now has been the fastest detailed placer publicly available. Furthermore, this speedup occurs while producing comparable or superior QoR. With these results, we speculate promising research directions towards scalable, high quality FPGA placement flows that can change the user experience from an overnight wait-time to a coffee break wait-time – even on large benchmarks.

## Categories and Subject Descriptors

B.7.2 [Hardware]: Design Aids - *Placement and routing*

## General Terms

Algorithms, Design, Performance

## Keywords

Bipartite Matching, Convex Optimization, FPGA, Quadratic Placement

## 1. INTRODUCTION

The FPGA industry has seen the steady growth of device capacity follow Moore’s law to the point that today many soft processors can be implemented on a single FPGA device. This enables FPGA market expansion to new territories, such as high performance computing (HPC). How-

ever, to deliver the user experience competitive to HPC substitutes such as general purpose graphic processing units (GPGPUs) and many-core processors, where compile time in minutes is the norm, the scalability of key CAD algorithms is emerging as a new priority.

Among the many challenges the community has to overcome, placement is one of the most important; not only because it accounts for the significant share of runtime in the complete compilation run, but also because of its critical impact on the quality of result (QoR).

For FPGAs, the most widely used placement paradigm has been simulated annealing due to its good QoR, as exemplified by the VPR [7]. However, even though multi-level [22] and parallelization[18] techniques have successfully reduced the runtime of annealing based placers, simulated annealing techniques pay significant runtime for good QoR. As a result, ASIC placers seem to have abandoned simulated annealing in favoring several other paradigms, which we refer to as *ASIC placement flows*:

- Multi-level partitioning based placement: using multi-level clustering and partitioning, such as Capo[8] and Feng Shui[15];
- Nonlinear analytical placement: using smooth approximation of placement objectives, and conjugate gradient nonlinear solver, such as APlace[14] and mPL[9];
- Scalable quadratic placement: using various force-directed formulations to quickly legalize a quadratic placement result, such as FastPlace[24].

FPGA capacity has reached the scale of their ASIC counterparts, which brings back an interesting old question: *How suitable are ASIC placement flows for FPGAs?* Although there have been prior attempts to adapt ASIC flows to FPGAs[26], results were reported only for small circuits with an order of 1K LUTs. These results are therefore inconclusive for modern devices that typically contain 10K-100K LUTs. Furthermore, the impact of recent advancements in ASIC placers, as seen in the successful ISPD placement contest, has yet to be quantified within an FPGA context.

We therefore revisit this issue in a modern setting using two directions:

- Q1 What is the exact QoR and runtime gap between ASIC placement flows and VPR?
- Q2 Can the QoR and runtime gap, if any, between ASIC placement flows and VPR be closed, while achieving speedup significant enough to change the user experience (i.e. order of magnitude reduction in wait time)?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA’10, February 21–23, 2010, Monterey, California, USA.  
Copyright 2010 ACM 978-1-60558-911-4/10/02 ...\$5.00.

Answering Q1 is not without difficulties: First, the ASIC and FPGA communities have been using different benchmark suites, in particular, there is no widely accepted, publicly available large FPGA placement benchmarks. Second, different frontend flows and device architectures result in netlists with different topological characteristics even for the same benchmark. Third, different QoR metrics were used: while academic ASIC placers all use half-perimeter wirelength (HPWL), the FPGA community regards timing-driven metric as more important and credible.

To draw useful conclusion on Q1, it is important to make an *apple-to-apple* comparison on *realistic* FPGA circuit benchmarks. We therefore make the following compromises: First, we adopted HPWL, since VPR can be easily modified to target HPWL without compromising optimality, whereas not all representative ASIC placers are available with modifiable source code to optimize for timing. Second, we constructed several benchmarks, following a procedure as rigorous as possible, to conduct the measurements. Although readers are recommended to take their own grain of salt due to the unavoidable compromises, surprising observations can be made.

- Partitioning-based and non-linear analytical placers (henceforth referred to as high-quality ASIC placers) achieve comparable QoR as VPR, however, high-quality ASIC placers have a modest runtime advantage (ranging from 1.6X-6X) which degrades as the cluster size increases. Furthermore, for a cluster size of 10, VPR is shown to be *faster* than many high-quality ASIC placement flows.
- Scalable quadratic placers achieves consistent speed up (ranging from 16X-40X), however, the QoR is not robust with respect to cluster size and availability of whitespace. Under the worse case of 0% whitespace and a cluster size of 10, the QoR ranges 21-29% worse than VPR. Under the nominal case of 10% white space and a cluster size of 10, the QoR ranges 5-20% worse than VPR.

To tackle Q2, we attempt to both improve QoR and reduce the runtime of ASIC placers by focusing on the detailed placement stage of the ASIC design flow. Detailed placement is relatively independent of the placement paradigm chosen, but occupies 30-74% of runtime of scalable placement and, as we will show, has a significant impact on the final placement quality. More specifically, we have developed a scalable matching-based detailed placement algorithm, called *MDP*. MDP achieves an over 3X to 7X speed up over the fastest academic detailed placer (FastDP), while achieving comparable or superior wirelength. We show that with MDP the wirelength gap between ASIC flows and the VPR flow can be dramatically reduced while significantly improving the runtime usage of the overall placement flow.

The rest of the paper is organized as follows. In Section 2, we detail the measurement methodology and results. In Section 3, we describe our proposed detailed placer. In Section 4, we provide results on the impact of MDP on ASIC placers, before we explore future directions on scalable placement for FPGAs.

## 2. MEASURING THE GAP

### 2.1 Methodology

To answer Q1, we follow an experimental methodology described below.

**Choosing ASIC Placers:** The set of ASIC placers used in this study consisted of leading representative academic placers from several categories. Capo, from U Michigan, is used as the partitioning-based placement flow; mPL, from UCLA, is used as the multilevel, nonlinear analytical placement flow; and FastPlace, from U Iowa, is used as the quadratic analytical placement flow. Note that there are a number of other academic placers that report comparable wirelength results but are not discussed here[23, 25, 2]. In each of the selected placers, we used the version that produces the best results at the time of study: mPL6, Capo 10.2 and FastPlace 1.0 with FastPlaceDP 3.0 (for our results, FastPlace 3.0, the latest of FastPlace, produced inferior result than FastPlace 1.0). All the ASIC placement flows we used follow two steps: a global placement step where approximate locations of cells are found (here slight cell overlap is allowed) followed by a detailed placement step where the legal position of a cell are found. We adapted each ASIC placer to perform FPGA placement by assuming all cells in the netlist represent FPGA clusters and are unit width and height. Also, we use VPR to define the minimum number of columns and rows in the FPGA architecture required to fit the given circuit when we want minimal whitespace.

**Choosing Metrics:** To quantify the quality of placement of both ASIC and FPGA placers, we use half-perimeter wirelength (HPWL) as the single optimization metric. Admittedly, HPWL is not the most widely used metric in FPGAs. However, this is a good starting point for two reasons: First, most of the available academic ASIC placers are HPWL-driven, thus reporting on HPWL provides a fair and credible result; Second, it could be argued that routability or timing-driven placement can be derived from HPWL-driven placement by properly weighting the nets. Therefore, for the purpose of evaluating placement flows, the choice of HPWL is valid.

**Choosing Benchmarks:** A challenge in evaluating placement scalability and QoR is the lack of large academic circuits for FPGAs. Thus, to mitigate this problem and provide credible results, we adopted two benchmark sets.

The first benchmark set, consisting of 18 circuits, is the IBM placement benchmark, which are commonly used in the ASIC community. The advantage of the IBM benchmark is their availability as a realistic, industry benchmark; as well as their size (ranging from 10K-200K placeable objects), which can evaluate scalability. To carry out this study, we preprocess it such that the all cells are of uniform size, including the macros, to mimic uniform FPGA clusters. The drawback with the IBM set is that since cells and macros are mapped directly to individual FPGA clusters, the netlist structure does not reflect what is typically seen in cluster based FPGA architectures.

The second benchmark class include several large circuits in IWLS and Altera’s QUIP benchmark suite[4]. When mapped to 4-LUTs using ABC[20], circuits in can reach over 250K LUTs. Following technology mapping, we cluster the netlist using t-vpack[7]. Circuits in this class are created from realistic designs and are synthesized using an FPGA design flow. In addition to these, we derive circuits from 15 medium and large sized circuits in the IWLS benchmark suite. To ensure that the size of these benchmarks matched the typical device size of modern FPGAs (order of 100K LUTs), we used a technique first devised at Altera[3]. Here, each benchmark core is replicated 5 to 20 times until their

circuit size reaches an order of 100K LUTs. Following replication, cores are connected together via long shift registers which connect to primary input and output pins as described in Altera’s literature[3]. Primary inputs and outputs are limited such that they follow Rent’s rule with a rent exponent of 0.5 and constant of 1.0 (e.g. a circuit with 100K LUTs would have  $100K^{0.5} = 316$  inputs and outputs [16]). Once created, these circuits were mapped and packed into clusters. A subset of the circuit sizes of our benchmark sets is shown in Table 1.

Circuit	4-LUTs	Circuit	Clusters (cells)
uoft	170069	ibm09	53110
netcard_S10	203750	ibm10	68685
b22_S10	90508	ibm11	70152
vga_lcd_S10	157055	ibm12	70439
des_perf_S10	168690	ibm13	83709
b19_l_S10	306719	ibm14	147088
b17_S10	124670	ibm15	161187
b18_S20	628956	ibm16	182980
vga_lcd_S20	627868	ibm17	184752
des_perf_S20	674260	ibm18	210341

**Table 1: Sample of circuits and their respective sizes used in the study. (Left) Circuits derived from the Quip and IWLS set, if suffixed with S10, the circuit was replicated 10 times, and with S20, the circuits were replicated 20 times. (Right) IBM circuit sets, where each cell is mapped directly into a FPGA cluster.**

Although all benchmarks are created from credible sources, each has its pros and cons. The readers are advised to judge the reported results with their own grain of salt.

**Impact of Clustering:** Modern FPGA architectures are often cluster based. It could be argued that cluster size may have an impact on the netlist topology. Since topology was reported by[17] to have an appreciable impact on the effectiveness of different placement flows, in our study we sweep the benchmark sets for cluster size ( $N$ ) of 1, 4, and 10 respectively. The exception to this was the IBM set since cells in the IBM set often cannot map to single output LUTs for clustering and instead cells are mapped directly to clusters.

**Impact of Whitespace:** Whitespace has a significant impact to the performance of analytical placers and QoR. Thus, we also sweep across three values of whitespace in our results: 0%, 10%, and 25%. Note that in the case of 0%, we attempt to minimize whitespace, although there may still exist a few empty cells, since not all circuits may fit perfectly in a  $x, y$  grid for a given the aspect ratio of 1 to 1.1.

## 2.2 Results

Using these benchmarks sets, we compared the ASIC placement flow to VPR. Due to the extreme sizes of our benchmark circuits, VPR was run in fast-mode (`-fast`), which significantly reduces the runtime of VPR at the cost of some QoR. If not run in fast-mode, VPR takes several days to complete the placement of a single circuit for the larger circuits.<sup>1</sup> Table 2 to 5 summarizes our results and shows a

<sup>1</sup>All experiments were run on a 64-bit Linux machine using an Intel Xeon quad core 1.6 GHz processor with 8GB of RAM. Without VPR fast-mode, each benchmark set takes several weeks to complete a single run.

comparison of VPR against each ASIC placer. Each table shows the final wirelength values in units of  $10^5$  ( $FWL$ ), the total placer runtime in seconds ( $TCPU$ ), and the detailed placement runtime in seconds ( $DPCPU$ ). In the final row of each table, we show the average ratio of VPR’s QoR over the ASIC flow’s QoR, the average ratio of VPR’s runtime over the ASIC flow’s runtime, and the average percentage of runtime taken by the detailed placer. These averages are taken with respect to all our circuits, however, only a subset of the circuits used are shown in detail. In all tables, we group the results with respect to 0, 10, and 25 percent whitespace, along with their associated averages. In cases where the placer failed to complete, we mark them with NA. When varying the cluster size,  $N$ , we restrict the number of inputs,  $I$ , to each cluster to  $I = 2N + 2$  [6]. For example, for a cluster of size of  $N = 4$ , we restricted the number of cluster inputs to 10.

The summarized results give a few clear observations. First, analytical placement, as shown by FastPlace, is significantly more scalable than VPR where FastPlace is consistently an order of magnitude faster than VPR. However, this runtime performance is paid for in QoR where VPR’s average final wirelength is approximately 50% to 20% better than FastPlace depending on the architecture and benchmark set used. This observation is different to what has been reported in previous studies [19], where FastPlace has been able to meet or exceed other analytical placers in terms of wirelength. This discrepancy in the results can be explained by two primary differences between our flow and the experiments presented in [19]: the available whitespace and the circuit topology resulting from FPGA clustering.

Ideally, setting whitespace to 0% is desired since this minimized the required FPGA area. However, our results show that this strict limitation results in poor performance in QoR, particularly for FastPlace. For mPL and FastPlace, whitespace has a significant impact to the resulting QoR. For VPR, the final wirelength changes very little with respect to whitespace, similarly for Capo. However, for FastPlace the final wirelength is significantly less when the available whitespace increases. In fact, mPL and FastPlace outperforms VPR both in terms of runtime and QoR in many cases when white space is set to 25% of the placement area.

In addition to whitespace, it appears that circuit topology due to cluster size has a significant impact on the QoR gap for FastPlace and Capo. This is not surprising since ASIC placers are not tuned for running on clustered netlists, which tend to pack local connections within a cluster. By packing local connections, the resulting netlist has several high-fanout global nets. An example of this is shown in Table 6 which shows the average fanout of the netlist with respect to the cluster size. Such a topology is known to lead to poor QoR in analytical placers and has been well studied [17].

$N$	1	4	10
$Ave_{fanout}$	3.47	13.9	34.6

**Table 6: Illustration of average fanout of a cluster ( $Ave_{fanout}$ ) with respect to the cluster size ( $N$ ) for circuit b19.**

One surprising observation is that despite the use of a multilevel paradigm, both mPL and Capo are not significantly

Circuit	Whitespace	VPR		Capo			mPL			FP		
		FWL	TCPU	FWL	TCPU	DPCPU	FWL	TCPU	DPCPU	FWL	TCPU	DPCPU
ibm10	0%	13.44	1029	13.09	907	272	14.04	446	227	15.32	55	32
ibm10	10%	13.58	1235	13.11	817	262	12.55	332	121	13.51	38	17
ibm10	25%	13.26	1252	13.43	750	251	12.78	370	135	13.34	40	16
ibm11	0%	11.70	1026	11.96	806	241	11.68	376	163	13.96	54	33
ibm11	10%	12.72	1033	12.08	861	272	11.25	318	118	12.56	36	15
ibm11	25%	11.41	1144	11.79	723	200	11.14	355	135	NA	NA	NA
ibm12	0%	16.15	1062	17.16	1404	704	16.42	438	182	18.42	60	33
ibm12	10%	16.36	1352	17.37	1374	728	16.02	372	128	16.58	51	21
ibm12	25%	17.08	1154	17.20	1306	689	15.98	408	144	16.67	49	17
ibm13	0%	14.10	1629	14.56	1149	323	15.77	578	276	16.84	64	37
ibm13	10%	14.17	1614	15.03	1061	335	13.73	438	148	14.68	49	23
ibm13	25%	14.07	1790	15.04	914	279	14.08	465	168	15.12	51	20
ibm14	0%	28.96	4093	26.70	2223	600	29.26	979	446	38.01	242	155
ibm14	10%	28.81	4223	27.29	2163	602	25.69	785	234	28.11	103	41
ibm14	25%	27.08	3837	27.10	1736	500	26.05	826	276	28.08	119	40
ibm15	0%	32.97	4913	33.92	2810	825	34.61	1071	415	58.64	365	276
ibm15	10%	35.86	4887	34.95	2627	777	32.60	963	289	36.53	153	69
ibm15	25%	35.07	5176	34.74	2121	638	33.33	1078	341	37.50	153	62
ibm16	0%	37.60	6419	36.70	3219	846	34.98	1194	455	53.52	358	237
ibm16	10%	36.22	6083	37.32	2761	828	34.38	1037	320	39.44	234	117
ibm16	25%	37.04	5075	37.06	2981	832	34.25	1201	330	37.29	155	58
ibm17	0%	44.72	6173	49.02	3499	895	51.09	1376	654	83.43	553	421
ibm17	10%	44.72	6173	49.04	3311	846	44.43	1057	343	70.22	411	309
ibm17	25%	45.84	5968	48.36	3254	781	45.09	1362	393	46.90	178	71
ibm18	0%	36.33	6186	35.36	3506	778	41.11	1617	831	62.38	538	429
ibm18	10%	37.11	5922	35.56	3226	796	32.81	1120	342	37.81	216	90
ibm18	25%	39.39	6728	36.03	2694	686	33.76	1345	423	37.74	189	76
Ave	0%			0.95	1.85	30%	0.87	3.38	45%	0.76	21.02	66%
Ave	10%			1.00	1.66	32%	1.08	4.64	33%	0.95	29.11	47%
Ave	25%			0.99	1.86	31%	1.05	4.04	33%	0.98	32.29	38%

Table 2: IBM circuit set (for runtime and QoR average, the larger the number the better the placer is with respect to VPR).

Circuit	Whitespace	VPR		Capo			mPL			FP		
		FWL	TCPU	FWL	TCPU	DPCPU	FWL	TCPU	DPCPU	FWL	TCPU	DPCPU
netcard_S10	0%	11.84	4069	9.21	1861	537	15.02	2927	893	24.21	162	105
netcard_S10	10%	11.97	3828	9.36	1420	477	11.24	2675	375	14.57	91	32
netcard_S10	25%	12.19	3581	9.11	1484	471	11.83	2913	494	13.96	84	23
uoft	0%	17.82	2997	11.22	1483	532	12.38	1249	364	20.41	219	138
uoft	10%	15.97	2959	11.31	1300	556	11.03	1114	218	16.56	140	54
uoft	25%	16.80	2873	11.34	1484	749	11.94	1638	304	17.22	135	52
b18_S10	0%	14.58	4133	12.04	1806	596	16.24	1679	919	17.84	219	114
b18_S10	10%	15.68	3533	12.31	1683	656	11.55	1036	306	15.56	184	74
b18_S10	25%	15.93	3317	12.65	1758	688	11.55	1143	377	14.73	193	85
b18_1_S10	0%	15.95	4488	12.17	1822	621	12.29	1306	513	20.32	281	166
b18_1_S10	10%	15.76	4428	12.29	1783	694	11.46	1037	307	14.61	186	77
b18_1_S10	25%	15.52	4057	12.45	1806	703	11.64	1167	384	14.97	178	74
b19_S20	0%	30.35	11921	22.71	3934	1298	30.42	3734	2195	47.03	712	452
b19_S20	10%	32.65	9514	23.16	3948	1379	21.74	2264	598	30.62	429	206
b19_S20	25%	32.52	9327	23.17	3697	1441	22.09	2731	738	29.50	437	199
b19_1_S20	0%	30.46	10969	22.60	4237	1232	NA	NA	NA	51.09	1111	555
b19_1_S20	10%	30.26	11264	23.32	3780	1357	21.87	2219	596	30.48	428	190
b19_1_S20	25%	31.15	10707	23.57	3661	1438	22.25	2480	731	29.26	442	188
b22_S10	0%	8.92	1815	6.78	886	309	8.25	964	588	9.77	125	69
b22_S10	10%	8.64	1653	7.03	904	392	6.51	569	177	8.42	95	41
b22_S10	25%	8.37	1709	6.90	916	399	6.61	664	219	8.19	100	40
b18_S20	0%	71.14	29268	53.78	9199	2160	82.09	6152	3203	NA	NA	NA
b18_S20	10%	74.20	32103	53.68	9663	3303	51.59	4178	1216	106.94	1484	952
b18_S20	25%	72.06	31290	54.27	9192	3228	52.80	4590	1517	76.27	1155	614
b18_1_S20	0%	76.98	29682	53.17	8750	2109	79.13	6876	3960	NA	NA	NA
b18_1_S20	10%	72.34	32466	53.93	9807	3687	51.59	4209	1284	103.93	1466	976
b18_1_S20	25%	77.26	27964	54.26	10880	4281	52.78	4810	1525	70.58	1080	545
Ave	0%			1.23	1.90	31%	1.01	1.18	49%	0.72	14.84	60%
Ave	10%			1.14	2.54	38%	1.06	4.32	28%	1.02	21.69	46%
Ave	25%			1.15	2.39	40%	1.13	3.59	30%	1.07	20.56	42%

Table 3: Quip and IWLS data set, for a cluster size of one 4-LUT ( $N=1$ , for runtime and QoR average, the larger the number the better the placer is with respect to VPR).

Circuit	Whitespace	VPR		Capo			mPL			FP		
		FWL	TCPU	FWL	TCPU	DPCPU	FWL	TCPU	DPCPU	FWL	TCPU	DPCPU
netcard_S10	0%	26.85	827	26.15	910	338	30.00	845	365	38.45	72	60
netcard_S10	10%	25.89	798	26.65	810	319	25.53	602	140	29.66	29	17
netcard_S10	25%	26.30	814	26.55	821	323	25.97	634	118	30.24	31	18
uoft	0%	9.42	543	8.35	505	217	7.70	284	104	10.55	41	24
uoft	10%	9.90	526	8.52	466	198	7.97	288	106	NA	NA	NA
uoft	25%	9.92	496	8.55	511	229	7.95	366	129	10.16	50	26
b18_S10	0%	13.54	1252	12.44	958	285	11.71	1315	210	16.54	76	49
b18_S10	10%	13.10	1238	12.34	872	288	11.55	1358	178	15.52	67	44
b18_S10	25%	13.53	1199	12.66	838	269	11.72	1376	201	14.11	67	42
b18_L_S10	0%	12.97	1176	12.41	936	282	11.87	1323	211	16.97	72	48
b18_L_S10	10%	13.32	1058	12.59	913	299	11.65	1172	179	15.12	64	42
b18_L_S10	25%	13.58	1073	12.80	853	274	11.74	1326	204	15.29	63	42
b19_S20	0%	25.82	3012	24.28	2008	543	29.98	3182	1083	40.01	259	206
b19_S20	10%	26.75	3145	24.23	1964	566	22.72	2693	343	29.79	128	80
b19_S20	25%	27.11	2636	24.28	1790	557	22.92	2922	388	28.53	128	78
b19_L_S20	0%	26.00	2653	24.27	2113	604	32.43	3506	1126	38.80	164	116
b19_L_S20	10%	26.50	2889	23.93	1818	542	22.71	2471	343	31.96	144	90
b19_L_S20	25%	27.33	2856	24.31	1743	527	22.96	2862	386	28.45	132	80
b22_S10	0%	7.79	410	8.15	494	165	7.63	630	125	10.20	39	27
b22_S10	10%	7.66	401	8.07	438	157	7.63	545	106	10.72	35	24
b22_S10	25%	7.73	396	8.11	452	158	7.62	647	113	9.51	33	22
b18_S20	0%	56.99	8726	52.11	4248	1064	51.44	5395	1152	85.68	368	252
b18_S20	10%	59.01	8056	52.62	3867	1054	49.20	5151	701	70.31	308	181
b18_S20	25%	59.48	7982	53.20	3797	1044	49.54	5913	790	64.45	306	186
b18_L_S20	0%	59.66	9449	52.32	4533	1193	58.29	6249	1486	94.49	430	320
b18_L_S20	10%	57.45	9378	52.69	3899	1102	49.24	5231	701	70.35	296	184
b18_L_S20	25%	56.17	8137	53.33	3747	1050	49.74	6005	802	66.67	302	179
Ave	0%			1.04	0.95	31%	0.94	0.91	26%	0.75	14.14	71%
Ave	10%			1.06	0.96	33%	1.13	1.19	18%	0.85	18.00	64%
Ave	25%			1.06	0.94	33%	1.13	1.03	17%	0.91	17.59	61%

Table 4: Quip and IWLS data set, for a cluster size of four 4-LUTs ( $N=4$ , for runtime and QoR average, the larger the number the better the placer is with respect to VPR).

Circuit	Whitespace	VPR		Capo			mPL			FP		
		FWL	TCPU	FWL	TCPU	DPCPU	FWL	TCPU	DPCPU	FWL	TCPU	DPCPU
netcard_S10	0%	17.71	322	18.50	668	383	20.11	709	294	23.73	21	15
netcard_S10	10%	17.81	322	18.70	630	380	17.88	555	130	23.26	20	14
netcard_S10	25%	18.00	330	18.54	597	360	17.57	591	138	20.27	13	7
uoft	0%	6.40	186	5.84	258	105	5.38	244	74	6.52	15	9
uoft	10%	6.13	179	5.96	258	122	5.43	234	77	6.71	14	8
uoft	25%	6.82	177	5.92	248	111	5.56	259	88	6.38	13	7
b18_S10	0%	9.84	553	9.35	709	294	10.40	1066	417	12.27	38	27
b18_S10	10%	10.13	507	9.31	681	278	8.75	862	167	12.37	37	27
b18_S10	25%	9.33	529	9.47	693	309	8.77	938	190	12.16	29	19
b18_L_S10	0%	9.81	441	9.31	699	290	10.61	1216	458	12.02	40	30
b18_L_S10	10%	9.70	414	9.31	640	287	8.80	869	169	11.95	36	25
b18_L_S10	25%	10.14	458	9.41	727	339	8.63	923	193	12.03	33	21
b19_S20	0%	19.71	1323	18.29	1628	868	NA	NA	NA	26.10	96	70
b19_S20	10%	19.30	1332	18.11	1420	689	17.06	1986	330	24.68	75	54
b19_S20	25%	20.29	1191	18.39	1595	826	17.15	2131	375	24.03	67	46
b19_L_S20	0%	20.99	1172	18.31	1580	831	23.48	2627	1127	26.85	75	55
b19_L_S20	10%	20.27	1294	18.18	1409	686	17.28	1938	330	26.21	87	63
b19_L_S20	25%	19.63	1265	18.56	1504	802	17.44	1946	386	24.19	65	40
b22_S10	0%	5.67	176	5.94	387	208	5.56	448	95	8.20	25	19
b22_S10	10%	5.81	211	5.93	366	167	5.49	465	102	7.63	20	14
b22_S10	25%	5.78	203	6.05	331	156	5.59	498	107	7.46	21	16
b18_S20	0%	39.53	3711	39.35	3956	1736	52.11	6024	2876	NA	NA	NA
b18_S20	10%	41.56	3625	39.17	3076	1101	36.21	4451	656	54.81	149	101
b18_S20	25%	41.52	3573	39.35	3189	1187	36.56	4385	761	52.38	118	69
b18_L_S20	0%	42.76	3610	38.73	3121	1125	53.90	6398	2859	58.51	160	119
b18_L_S20	10%	41.68	4135	39.11	3042	1159	36.51	4475	671	57.23	165	111
b18_L_S20	25%	41.65	4209	39.67	3147	1205	36.71	4343	769	54.28	147	95
Ave	0%			1.01	0.53	47%	0.89	0.41	34%	0.77	11.70	72%
Ave	10%			1.03	0.60	44%	1.10	0.62	20%	0.79	13.27	69%
Ave	25%			1.04	0.58	46%	1.11	0.59	21%	0.80	18.50	64%

Table 5: Quip and IWLS data set, for a cluster size of ten 4-LUTs ( $N=10$ , for runtime and QoR average, the larger the number the better the placer is with respect to VPR).

faster and in some cases even slower than VPR’s simulated annealing based placement. This contradicts intuition since one of the primary reasons simulated annealing was abandoned for ASIC placement was due to scalability concerns. On the other hand, Capo and mPL are quite competitive in terms of QoR with VPR where Capo can even outperform VPR in QoR for many of the benchmark sets.

The previous observations lead to several conclusion. First, if given enough whitespace, ASIC placement flows are competitive with VPR. For example, mPL does extremely well, and even beats VPR in terms of WL if given enough whitespace. However, there are many cases where such a large overhead is not acceptable, thus, the QoR gap of ASIC placement flows with respect to VPR for these cases is still a concern. Our results exposes areas which need to be improved in order for ASIC placement flow to be successfully applied to FPGAs. This leads us to the second part of our study: Can we close the gap such that ASIC placement flows can be widely used in an FPGA context?

### 3. MDP: MATCHING BASED DETAILED PLACER

In this section, we attempt to improve the runtime and QoR for detailed placement. This is justified by noting that in our measurement, detailed placement occupies often over half of the runtime of FastPlace+FastDP flow, which show the best promise as a scalable placement flow. In addition, it is well known that detailed placement has a large impact on the final placement quality [21].

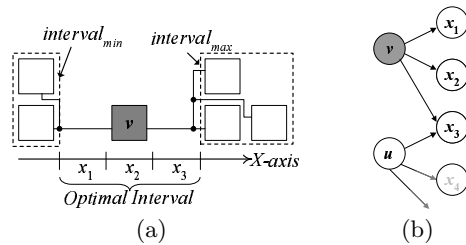
We developed a scalable matching-based detailed placement algorithm, called *MDP*, that both achieves an order of magnitude speed up and can produce significantly superior QoR over most detailed placers in leading academic placers. Another benefit of MDP is that these results are independent of the global placement algorithm used, and as a result, MDP works well in conjunction with all the existing global placement algorithms that we present here.

#### 3.1 Background

Detailed placement can be classified into three main categories. The first class slices the two-dimensional placement area into small windows and computes an optimal cell ordering inside the window using branch-and-bound [8] and/or dynamic programming [14]. However, due to the computational complexity the window size is very small, which severely limits the optimization coverage.

The second class, represented by Domino [11], managed to enlarge the window by formulating a transportation problem (essentially a min-cost bipartite matching problem) solved by the augmenting path algorithm. However, to guarantee a perfect matching, a fully connected bipartite graph is constructed in each window which considerably slows down the runtime. To improve this, a trial-and-error edge-prune heuristic is attempted in [1], where it uses *maximum-bipartite matching* to repeatedly check if a pruned graph stays feasible. However, the benefit of this is not fully leveraged due to the trial-and-error iterations resulting from a lack of knowledge in its edge-prune heuristic.

The third class exploits the *optimal interval* which defines a placement region for a given cell that results in the minimum HPWL cost for all nets attached to the cell, assuming all other cells remain stationary. An example of a one-dimensional optimal interval is shown in Figure 1. Here,

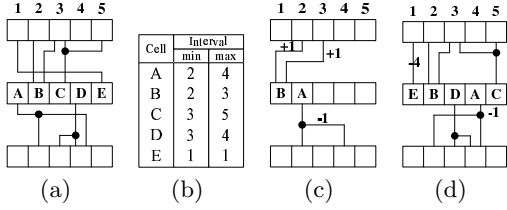


**Figure 1: An illustration of an *Optimal Interval* along the *x*-axis and the translation of the cells to a bipartite graph. In (b) the nodes on the left are the derived from the placeable nodes (i.e. clusters) and are referred to as *Supply Nodes*; while the nodes on the right are derived from the locations along a given axis, and are referred to as *Demand Nodes*.**

the optimal interval of the cell,  $v$ , is defined as the interval between the median values of net bounding box values along a given axis. The bounding boxes are defined assuming cell  $v$  has been removed from each net. We show this in Figure 1(a) where we show the bounding boxes for the nets connected to node  $v$  that create four vertical lines. The two median  $x$  values of these lines create an optimal interval in the  $x$ -axis as illustrated in the previous figure. The optimal interval usually contains multiple locations, which offers flexibility to place a cell optimally. If generalized to a 2D region, it is called a *optimal region*.

Existing placers leverage the optimal interval through pairwise swaps, where cell pairs are swapped in a fashion that attempts to increase the number of cells placed within their optimal interval. In FastPlace, FastDP [21] swaps a cell,  $v$ , with another cell,  $u$ , in its optimal region — all the cells in the optimal region of  $v$  are evaluated and the best swap reducing the wirelength is chosen for a single swap with  $v$ .

mPL applies a “chained” set of swaps in an attempt to place more cells into their optimal region. In this approach, given a starting cell  $A$ , a second cell  $B$  is randomly chosen from  $A$ ’s optimal region, and a third cell  $C$  is randomly chosen from  $B$ ’s optimal region in the same manner, and so on until five cells are chosen. The interchange is accepted if the best permutation of the five cells reduces wirelength. Since both the single swap method in FastDP and chained method in mPL are greedy, the optimal arrangement is often missed. Figure 2 gives an example of this scenario when applied to a 1D optimal interval. Here we show three rows in a placement region. The middle row shows five cells in detail, along with their net connections to cells in the two adjacent rows. Figure 2(a) shows an initial placement for the five cells in the middle row, whose  $x$ -axis minimum and maximum values of the optimal intervals are summarized in Figure 2(b). Using the pair-wise swap, cell  $A$  is attempted to swap with  $B$  as shown in Figure 2(c). This results in an increase in wirelength of +1 for each net connected to cell  $B$ , and a decrease in wirelength of -1 to the net connected to cell  $A$ . Since this results in an overall increase in wirelength of +1, the swap is rejected. Next,  $A$  is swapped with  $C$  and evaluated, and so on. In this example, none of the pair-wise swaps with  $A$  reduce wirelength, and thus  $A$  gets stuck at the current location and cannot be placed into its optimal interval. This occurs even though there exists a permutation of the cells such that all of the cells are placed into their



**Figure 2: (a) Starting Placement (d) Optimal Interval Based Reordering**

optimal interval as shown in Figure 2(d) (total wirelength reduction is -5).

### 3.2 Algorithm Overview

The previous example highlights how greedy based swapping methods can get trapped in local minima. As an improvement, we prove that the detailed placement problem of moving cells to their optimal interval is exactly the *maximum-bipartite matching* problem where we are attempting to match cells to a location found in their optimal interval. By using this insight, we are able to solve the detailed placement problem globally, which avoids the problems associated with single swap and chained swap approaches.

The maximum-bipartite matching problem attempts to match *supply nodes* to their associated *demand nodes*. When applied to detailed placement, our supply nodes represent cells, demand nodes represent locations, and edges connect cells to locations within the optimal interval. This is highlighted in Figure 1(b). If applied to FPGAs, we can assume each cell represents a cluster of unit width and each  $(x,y)$  location can place at most one cluster.

Using these insights, we propose a scalable matching-based FPGA detailed placement called MDP which leverages three primary innovations. Firstly, MDP formulates a *maximum-bipartite matching* problem between clusters and their optimal intervals to maximize the number of clusters moved to their optimal locations in a single move. In contrast to the pair-wise or the chain-wise movement which are often restricted to less than 10 clusters, this formulation allows *simultaneous* resolution of clusters spanning the entire strip of a placement region which can contain *thousands* of clusters. Secondly, MDP speeds up the maximum matching solver to  $O(n \log n)$  time, while maintaining its exact optimality, by exploiting a special property of optimal intervals. Thirdly, MDP resolves the residual unmatched clusters and locations with a min-cost matching algorithm, however, with a significantly reduced problem size. The result achieved is an overall near-optimal solution that leverages a single *simultaneous move* for all clusters in the entire strip of a placement region.

Our generic flow for detailed placement consists of three phases: (i) legalization, (ii) matching-based cluster reordering, and (iii) greedy swapping of neighbouring clusters. The *legalization phase* transforms the global placement result into an overlap-free layout using a similar approach as [13] where cells are shifted into a legal location based on their order in the  $x$  and  $y$  axis. This is extremely fast but on its own leads to poor QoR. The *cluster reordering* step aims to improve wirelength by moving the placed clusters. This step is the primary contribution of the new MDP algorithm. The result is further optimized by a fast *greedy swapping* of

neighbouring clusters where swapping of clusters only occurs if it results in reduced wirelength.

---

#### Algorithm 3.1: The MDP Algorithm

---

```

1 repeat
2   Perform AxisCellReordering( $x$ -axis);
3   Perform AxisCellReordering( $y$ -axis);
4 until no significant improvement in wirelength ;

Procedure: AxisCellReordering( $axis$ )
1 Slice the placement area along  $axis$ ;
2 foreach slice  $S$  do
3   Get the Optimal Interval for all clusters in  $S$ ;
4   Perform Interval Bipartite Matching;
5   if  $Slice$  has unmatched clusters then
6     Perform MincostGraphConstruction;
7     Solve min-cost matching on the graphs;

Procedure: MincostGraphConstruction
1 Partition nodes into separate graphs;
2 foreach identified partitioned graph do
3   Perform Perfect Matching Construction;
4   Perform Optimal Relaxation;

```

---

The cluster reordering step is highlighted in Algorithm 3.1. The core of MDP’s matching-based cluster reordering is composed of an iterative procedure in which the wirelength optimizations along rows ( $x$ -axis) or columns ( $y$ -axis) are interleaved, as shown in the first 4 lines of Algorithm 3.1. When calculating wirelength, we adopted a net model proposed in Domino to estimate the half perimeter wirelength (HPWL) and calculate the optimal interval. This model provides an accurate estimation “in the neighbourhood of the current placement” [11]. Our study shows 95% of clusters, on average, are within at most five placement units of their optimal intervals in the global placement results, which justifies our formulation.

During the iterative procedure, the placement area is divided into slices consisting of two adjacent rows (columns) and the placement is improved by processing the successive row (column) slices until all rows (columns) are processed. The interleaved row and column-wise optimization is repeated until there is no significant wirelength reduction obtained.

In each slice, cluster reordering is performed by four steps (lines 3 to 7 in AxisCellReordering procedure of Algorithm 3.1). First, the optimal intervals are calculated for all the clusters in a slice. Second, the clusters and optimal intervals are used to form a maximum-bipartite matching problem as illustrated in Figure 1(b). Here, clusters act as supply nodes on the left and locations act as demand nodes on the right. An edge exists between a supply node and demand node if the location demand node is part of the optimal interval of the cluster supply node. Once constructed, we use a new  $O(n \log n)$  algorithm that exploits optimal intervals to solve the maximum-bipartite matching problem. If the matching result leads to a perfect matching, it automatically results in a minimum cost feasible matching where all clusters are placed at their matching demand node locations. Otherwise, the unmatched nodes (clusters and positions) are partitioned into disjoint graphs (MincostGraphConstruction procedure in Algorithm 3.1), and each graph is solved by a min-cost matching algorithm to ensure an overall legal and min-cost solution. These min-cost graphs are constructed with a small number of edges through a two-step process consisting of a one-to-one perfect matching construction to

provide a good upper bound of optimality followed by an edge addition step where low cost edges are added to connect supply nodes to more demand nodes.

### 3.3 Interval-Bipartite Matching

Since optimal intervals are finite, a special feature of our bipartite graph is that the demand nodes connected to a supply node create an interval, called a *demand interval*. Exploiting this feature significantly reduces the computational complexity of solving the maximum-bipartite matching algorithm.

For a bipartite graph with the  $n$  supply nodes and  $m$  edges, each matching can be found in  $O(m)$  time, leading to an overall  $O(mn)$  runtime. However, the interval property of the demand nodes implies the number of edges explicitly examined can be far less. If we process the supply nodes in a certain order, it is possible to pick a matching assignment for each supply node that will not needlessly block locations for the subsequent supply nodes. Thus, the matched demand node for a supply node can be found in  $O(1)$  time. This algorithm is shown in algorithm 3.2.

The order is based on their optimal regions where supply nodes are sorted according to their demand intervals in a lexicographical order on the 2-tuple  $(interval_{max}, -interval_{min})$ . That is, the demand intervals are sorted in ascending order according to their  $interval_{max}$  value; within that order, intervals with the same maximum coordinates are sorted in descending order according to their  $interval_{min}$  value. Then, the exact maximum matching result can be obtained in linear time by simply picking the minimum available demand node for each supply node.

---

#### Algorithm 3.2: The Interval Bipartite Matching

---

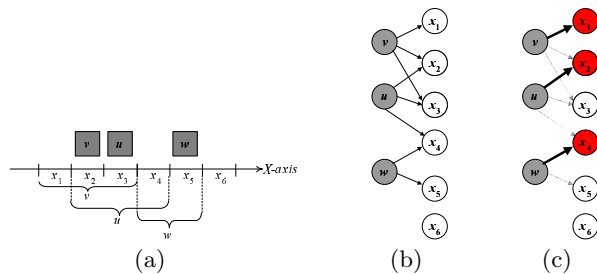
**Input:** A bipartite graph with  $n$  supply nodes  $S_1, \dots, S_n$  with demand intervals.

**Output:** Maximum matching of the bipartite graph.

- 1 Sort the demand intervals in lexicographical order on  $(interval_{max}, -interval_{min})$ ;
  - 2 Mark all the demand nodes as available;
  - 3 **foreach** supply node in this sorted order  $S_i$  **do**
  - 4     matched  $\leftarrow$  FALSE;
  - 5     **foreach** demand node  $D_j$  in the demand interval of  $S_i$  **do**
  - 6         **if**  $D_j$  is available **then**
  - 7              $S_i$  is matched to  $D_j$ ;
  - 8             Mark  $D_j$  as unavailable;
  - 9             matched  $\leftarrow$  TRUE;
  - 10         **break**;
  - 11     **if** matched = FALSE **then**
  - 12          $S_i$  becomes an unmatched node;
- 

An example of this process is illustrated in Figure 3. In Figure 3(a), three cells are shown, along with their  $x$ -axis optimal intervals shown by the horizontal braces. In Figure 3(b), the original circuit is converted into its associated bipartite graph. In the bipartite graph, supply nodes on the left, which represent clusters, have an edge to demand nodes on the right, which represent locations. Edges only connect clusters to locations within their optimal interval. Following this, Algorithm 3.2 is applied resulting in the maximal matching shown in Figure 3(c).

**THEOREM 3.1.** *The Interval-Bipartite Matching algorithm shown in Algorithm 3.2 finds the maximum-bipartite matching.*



**Figure 3: Illustration of matching process. (a) Original circuit, the optimal interval for each cluster is shown below the  $X$ -axis. (b) Conversion of circuit to a bipartite graph (c) Matching of supply nodes to demand nodes, resulting in maximal matching.**

**Proof:** Proof by induction on  $n$ , the number of the supply nodes. If  $n = 1$ , there is only one supply node and one demand interval and the theorem is obviously true. Now assume that the theorem is true for  $n - 1$  supply nodes. We will show that the theorem is true for  $n$  supply nodes.

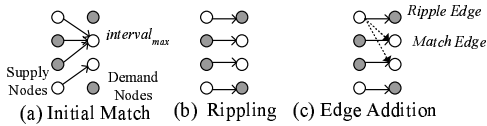
If there is an available demand node for  $S_n$ , the theorem is obviously true. Otherwise, two scenarios are possible: (i) if there is no other free demand nodes for  $S_1, \dots, S_{n-1}$ , the existing matching must be maximum for  $S_1, \dots, S_n$  because all the demand nodes are matched; (ii) if there is a free demand node,  $D_{free}$ , for  $S_i (1 \leq i \leq n - 1)$  which is matched to  $D_{match}$ , given the order,  $D_{match}$  must not be included in the demand interval of  $S_n$ . Hence, moving  $S_i$  to  $D_{free}$  will not solve the conflict for  $S_n$ . The existing matching is maximum for  $S_1, \dots, S_n$ .  $\square$

**Complexity:** The time complexity for the interval-bipartite matching problem is dominated by the sorting step as shown on line 1 in algorithm 3.2. As a result, the complexity of the interval-bipartite matching problem is  $O(n \log n)$ . This is significantly faster than the fastest known generic algorithm [10] which has a complexity of  $O(\sqrt{nm} \log(n^2/m) / \log n)$ . Even for sparse graphs, where  $m$  is small, we find our algorithm in practice is twice as fast as the generic algorithm in [10] when applied to our benchmark sets.

### 3.4 Min-cost Bipartite Matching

The maximum matching result does not necessarily lead to a perfect matching: some clusters are unmatched to a location. This will occur if there is a lot of overlap in a given region along an axis. One can think of this as a “congestion map” showing which locations are contended and which locations are free. The problem then steers to how to spread the clusters in congested areas towards the free locations while achieving an overall min-cost assignment. This can be naturally solved by a min-cost bipartite matching problem. In order to avoid solving the min-cost bipartite matching problem on the entire graph, we only run it on the existing set of unmatched nodes and a very small number of matched nodes. Furthermore, we partition this existing set of nodes into sub-graphs.

In order to place unmatched nodes we first match unmatched supply nodes to their  $interval_{max}$  location. This location may be common to several unmatched nodes, or already be occupied as shown in Figure 4(a) (in Figure 4, after running Algorithm 3.2, matched nodes are white and



**Figure 4: Sub-graph construction, grey nodes are unmatched.**

unmatched nodes are grey). Following this, we partition the graph into sub-graphs where we ensure that the number of demand and supply nodes are equivalent. If we order our demand nodes based on their axis value (i.e. if along the  $x$ -axis, the smallest  $x$  value is placed on the top of the graph), we partition the graph starting at a unmatched demand node (i.e. a free location). The partition is grown by adding the next demand node in the ordered sequence. Supply nodes are added to the partition if they are matched to a demand node that has been added to the partition. This process continues until the number of supply nodes is equivalent to the number of demand nodes, and the last demand node added to the partition is unmatched. Although in the worse case, this could add the entire bipartite graph to the partition, empirically we find that this creates graphs which typically contain no more than 30 nodes. By ending and beginning the partition at unmatched demand nodes, we ensure that the partition will contain at least one demand node that is part of the optimal interval for each supply node. Also, there exists a convex relationship between the HPWL and the cell position with respect to its optimal interval (i.e. as cells move farther from their optimal interval, their HPWL value increases monotonically). These points ensure that the partitioning scheme does not remove the potential of an optimal matching from its solution space and allows the min-cost matching solver find this solution.

Once the partition is created, we *ripple* out conflicting node matches as illustrated in Figure 4(b). Here, we order the supply nodes and demand nodes based on their current location followed by a pair-wise matching from the top to the bottom of the graph where matching edges are not allowed to “cross” over each other.

Once a one-to-one matching is made, additional edges are added to each supply node to expand the search space for matching supply nodes as shown in Figure 4(c). Here we show added edges for the top node only. A heuristic is used to limit the number of additional edges. If we define <sup>2</sup>  $\Delta cost = cost_{edge} - cost_{matched}$ , we add all edges with a  $\Delta cost = 0$ , three edges with  $\Delta cost = 1$ , one edge with  $\Delta cost = 2$ , and no edges with a  $\Delta cost > 2$ . Notice that the ripple edges may already include some of these edges, therefore a very small number of edges suffices to realize an almost-optimal solution (a mere 0.5% from the optimal solution is observed in our experiments).

Once the min-cost bipartite graph is constructed, we use a public domain cost scaling algorithm, CSA [12], to solve the min-cost matching problem.

## 4. EXPERIMENTS AND RESULTS

The goal of our experiments is two-fold: (i) to evaluate

<sup>2</sup>We are assuming  $cost_{match}$  is the minimum cost edge for a given supply node. This is true since the original matched node is  $interval_{max}$  which is within the optimal interval of the supply node.

Set	WP	TCPU $\frac{VPR}{ASIC}$		DPCPU	
		FastPlace		FastPlace	
		Native	MDP	%MDP	$\frac{FastDP}{MDP}$
IBM	0%	21.02	42.73	26.32%	5.22
IWLS/QUIP N=1	0%	14.84	29.73	17.51%	7.01
	10%	21.69	31.12	17.63%	3.44
	25%	20.56	29.38	17.14%	3.52
IWLS/QUIP N=4	0%	14.14	32.84	31.53%	5.38
	10%	18.00	34.84	31.95%	3.98
	25%	17.59	33.05	30.59%	3.91
IWLS/QUIP N=10	0%	11.70	25.78	35.96%	3.13
	10%	13.27	24.21	36.34%	3.20
	25%	18.50	30.57	34.52%	3.48

**Table 7: Runtime impact of MDP on FastPlace**

MDP’s<sup>3</sup> impact on runtime on the ASIC placement flow (ii) to quantify how well MDP narrows the QoR gap against VPR when used in conjunction with existing ASIC global placers. To explore the impact of MDP for both of these issues, we compare against FastPlace since it is shown to be the fastest publicly available placer for our data set. Furthermore, it is the only placer which can achieve the order of magnitude reduction in runtime with respect to VPR as shown in Section 2. Here, we first run FastPlace in an FPGA context and compare the runtime and QoR numbers against VPR. Next, we replace the detailed placer in FastPlace with MDP and record the overall runtime and QoR impact against VPR. Note that when evaluating MDP, we only show summarized results due to space limitations. For detailed results, please refer to [5].

**Runtime:** Table 7 summarizes the runtime results. Here we list the benchmark name in the first column along with the cluster size if applicable, followed by the percent whitespace ( $WP$ ) used in the FPGA architecture. The first set of columns headed by  $TCPU \frac{VPR}{ASIC}$  shows the average ratio of VPR’s total runtime over FastPlace’s total runtime. This contains two sets of data: the first column, *Native*, shows the runtime comparison of the native placement flow; the second column, *MDP*, shows the runtime comparison when MDP replaces the detailed placer in FastPlace. In the last two columns heading by *DPCPU*, we show the fraction of runtime occupied by MDP in the overall ASIC placement flow (%*MDP*) and compare the FastPlace detailed placement runtime with MDP’s runtime ( $\frac{FastDP}{MDP}$ ).

**QoR:** Table 8 summarizes MDP’s impact on QoR and first lists the benchmark set along with the cluster size, followed by the percent whitespace. Following this, we show the average ratio of wirelength produced by VPR over the wirelength produced by FastPlace. The wirelength comparison has two columns: the first column, *Native*, shows the ratio when the FastPlace detailed placer is used; and the second column, *MDP*, shows the average ratio when MDP is used instead of the FastPlace detailed placer.

**MDP Impact:** Looking at the runtime summary in Table 7, it is clear that MDP has a significant runtime advantage over the FastPlace detailed placer. Specifically, MDP is approximately 3X to 7X faster than the FastPlace detailed placer. As a result of this speedup, MDP occupies a much smaller amount of runtime in each overall placement flow.

When analyzing the QoR results, MDP consistently im-

<sup>3</sup>MDP is implemented in C and all experiments were run on a 64-bit Linux machine using an Intel Xeon quad core 1.6 GHz processor with 8GB of RAM.

Bench	WP	FP	
		Native	MDP
IBM	0%	0.76	0.94
IWLS/QUIP N=1	0%	0.72	1.03
	10%	1.02	1.03
	25%	1.07	1.08
IWLS/QUIP N=4	0%	0.75	0.87
	10%	0.85	0.94
	25%	0.91	0.94
IWLS/QUIP N=10	0%	0.77	0.87
	10%	0.79	0.92
	25%	0.80	0.90

**Table 8: QoR impact of MDP on FastPlace**

proves overall QoR when used in conjunction with FastPlace. Here, MDP’s improvement is dramatic: the wirelength improvement ranges from 12% to 31%.

Finally, when looking at the impact of clustering and whitespace on QoR, it appears that MDP mitigates clustering and whitespace issues found in FastPlace.

## 5. CONCLUSION AND FUTURE WORK

In the beginning of this paper, we raised the question: *How suitable are ASIC placement flows for FPGAs?*

One key finding of our study is that contrasting to what one would hope, the recent, best-in-class academic placers, with the exception of FastPlace, are not significantly faster than VPR. In particular, when the number of clusters in the netlist is less than 50K, VPR can outperform Capo both in terms of runtime and QoR<sup>4</sup>. Coupled with the trend of increasing cluster sizes in FPGAs, this may imply that simulated annealing based placement would still be in dominant use for a few more device generations.

The second key finding is that combining recent quadratic placers with improved detailed placement is a promising scalable placement flow for FPGAs. Take the FastPlace+MDP placement flow as an example: when run on the largest IBM benchmark, which contains 200K clusters, it takes only 4 minutes to complete, whereas VPR takes 5 hours, which dramatically change the user experience from overnight wait to coffee-time wait. Also, given ample white space, FastPlace+MDP can produce comparable HPWL result as VPR.

These findings naturally point to future research directions: further improving the robustness of scalable global placement with respect to cluster size and white space, and migrating the ASIC placement flows to optimize other metrics that are important to FPGAs, beyond HPWL.

## 6. ACKNOWLEDGEMENTS

The authors would like to thank the financial support from Altera, and National Sciences and Engineering Research Council of Canada.

## 7. REFERENCES

- [1] A. R. Agnihotri and P. H. Madden. Fast analytic placement using minimum cost flow. In *ASPDAC*, pages 128–134, January 2007.
- [2] C. Alpert, A. Kahng, G.-J. Nam, S. Reda, and P. Villarrubia. A semi-persistent clustering technique for VLSI circuit placement. In *ISPD*, pages 200–207, 2005.
- [3] Altera. OpenCore stamping and benchmarking methodology. Technical Report TB-098-1.1, Altera, 2008.
- [4] Altera Corporation. *Quartus II University Interface Program*.
- [5] H. Bian. Placement by marriage. M.A.sc. thesis, University of Toronto, Toronto, Canada, June 2008.
- [6] V. Betz and J. Rose. Cluster-based logic blocks for FPGAs: Area-efficiency vs. input sharing and size. In *CICC*, pages 551–554, 1997.
- [7] V. Betz and J. Rose. VPR : A new packing, placement and routing tool for FPGA research. In *FPL*, September 1997.
- [8] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Optimal partitioners and end-case placers for standard-cell layout. In *ISPD*, pages 90–96, 1999.
- [9] T. F. Chan, J. Cong, T. Kong, and J. R. Shinnerl. Multilevel optimization for large-scale circuit placement. In *ICCAD*, pages 171–176, November 2000.
- [10] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, and J. Stolfi. Augment or push? a computational study of bipartite. matching and unit capacity flow algorithms. Technical Report TR-98-036R, NEC, March 1998.
- [11] K. Doll, F. Johannes, and K. Anreich. Iterative placement improvement by network flow methods. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1189–1199, 1994.
- [12] A. V. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Mathematical Programming*, 71(2):153–177, December 1995.
- [13] D. Hill. Method and system for high speed detailed placement of cells within an integrated circuit design. In *US Patent*, number 6,370,673, 2002.
- [14] A. B. Kahng, S. Reda, and Q. Wang. Architecture and details of a high quality, large-scale analytical placer. In *ICCAD*, pages 890–897, November 2005.
- [15] A. Khatkate, C. Li, A. R. Agnihotri, M. C. Yildiz, S. Ono, C.-K. Koh, and P. H. Madden. Recursive bisection based mixed block placement. In *ISPD*, 2004.
- [16] B. Landman and R. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Trans. Computers*, C-20(12):1469 – 1479, December 1971.
- [17] Q. Liu and M. Marek-Sadowska. A study of netlist structure and placement efficiency. In *ISPD*, pages 198–203, 2004.
- [18] A. Ludwin, V. Betz, and K. Padalia. High-quality, deterministic parallel placement for FPGAs on commodity hardware. In *FPGA*, pages 14–23, 2008.
- [19] N. V. Min Pan and C. Chu. An efficient and effective detailed placement algorithm. In *ICCAD*, pages 48–55, Nov. 2005.
- [20] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG Rewriting: A fresh look at combinational logic synthesis. In *DAC*, pages 532–536, 2006.
- [21] M. Pan, N. Viswanathan, and C. Chu. An efficient and effective detailed placement algorithm. In *ICCAD*, pages 48–55, 2005.
- [22] Y. Sankar and J. Rose. Trading quality for compile time: Ultra-fast placement for FPGAs. In *FPGA*, pages 157–166, 1999.
- [23] T. Taghavi, X. Yang, and B.-K. Choi. Dragon2005: large-scale mixed-size placement tool. In *ISPD*, 2005.
- [24] N. Viswanathan and C. Chu. FastPlace: Efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model. In *ISPD*, April 2004.
- [25] Z. Xiu and R. A. Rutenbar. Mixed-size placement with fixed macrocells using grid-warping. In *ISPD*, 2007.
- [26] Y. Xu and M. Khalid. QPF: efficient quadratic placement for FPGAs. In *FPL*, Aug. 2005.

<sup>4</sup>Modern architectures contain on the order of 10K clusters[3].