SHARED RECONFIGURABLE ARCHITECTURES FOR CMPS

Matthew A. Watkins, Mark J. Cianchetti, and David H. Albonesi
Computer Systems Laboratory
Cornell University

ABSTRACT

This paper investigates reconfigurable architectures suitable for chip multiprocessors (CMPs). Prior research has established that augmenting a conventional processor with reconfigurable logic can dramatically improve the performance of certain application classes, but this comes at non-trivial power and area costs. Given substantial observed time and space differences in fabric usage, we propose that pools of programmable logic should be shared among multiple cores. While a common shared pool is more compact and power efficient, fabric conflicts may lead to large performance losses relative to per-core private fabrics.

We identify particular characteristics of past reconfigurable fabric designs that are particularly amenable to fabric sharing. We then propose *spatially* and *temporally* shared fabrics in a CMP. The sharing policies that we devise incur negligible performance loss compared to private fabrics, while cutting the area and peak power of the fabric by 4X.

1. INTRODUCTION

The microprocessor industry is rapidly transitioning to chip multiprocessors comprised of many processing cores. While the move away from single complex cores is indisputable, the most profitable way to organize future multicore chips is a topic of ongoing debate.

One attractive approach is to intermix conventional processing cores with reconfigurable logic that can be programmed to assume multiple specialized functions. This approach provides a form of *reconfigurable heterogeneity* that can be matched to changing workload characteristics at runtime. While the physical design may be homogeneous, with identical cores and attached programmable logic fabrics, each fabric can be configured at runtime to perform a different function according to the tasks assigned to the cores, or in the case of homogeneous threads from a parallel application, they may be configured identically.

Despite its potential, reconfigurable logic as an attached customizable unit has not yet been widely embraced by mainstream microprocessor manufacturers. One major impediment is the large power and area costs of FPGA technology relative to the fixed functionality of commercial microprocessors [1]. While researchers have made significant progress in devising *specialized programmable logic (SPL)* to bridge this gap, many microprocessor architects still view the costs as too high to justify their mainstream adoption.

In this paper, we propose shared SPL microarchitectures and low-level hardware control for future CMPs. In a multicore system where each core is coupled with its own fabric there are inevitably periods where one fabric is highly utilized while another lies largely or even completely idle. Thus, by *sharing* SPL fabric resources among multiple processor cores, programmable logic can be much more efficiently integrated – at far less power and area cost – into future multicore microprocessors.

In our approach, a number of standard cores share a common pool of SPL. Depending on the particular needs at any given point in time, each pool is dynamically partitioned among the cores, either *spatially*, where the shared fabric is physically partitioned among multiple cores, or *temporally*, where the fabric is shared in a time multiplexed manner. We develop spatial and temporal control policies that direct the cores to particular SPL partitions, or pipeline time slots, based on runtime statistics. We show in Section 5 that pooled SPL configurations guided by effective control policies have little impact on performance for both parallel and coarse-grain multithreaded workloads – compared to private SPL attached to each core – while significantly reducing SPL area and energy costs.

2. RELATED WORK

Several survey papers (e.g., [2, 3]) provide an overview of the contributions of prior reconfigurable computing projects. We focus on those proposals that address the low level SPL design, the SPL characteristics of prior approaches that we find particularly amenable to shared fabrics, and the integration of the fabric with a CMP.

PRISC [4], Proteus [5], Stretch [6], Chimaera [7], and DPGA [8] tightly integrate the fabric with the processor as a specializable execution unit. The fabric predominates in DISC [9] and NAPA [10] with the processor serving largely to feed the reconfigurable hardware. Garp [11, 12] and PipeRench [13, 14, 15] fall in between.

Other efforts [5, 16] have evaluated the high level integration of the fabric with a general purpose processor and overall system performance, but none have explored integrating a reconfigurable fabric in a multicore chip. Several efforts, however, have developed reconfigurable fabrics whose characteristics – in particular *higher computation granularity*, *row based design*, and *virtualization* – we found to be highly amenable to efficient SPL sharing.

Garp [12] and PipeRench [14] use two and eight bits, respectively, as the smallest computation granularity. Using larger granularities reduces power, area, and configuration bits at the cost of flexibility and density in design mapping. Most general purpose applications, however, do not require bit level manipulation and so the savings tend to outweigh the costs. While this does not directly ease sharing *per se*, it does significantly reduce area and power.

Row based reconfiguration is employed by a number of designs [7, 9, 14] for several reasons. The cycle time of the fabric is set by the row delay and thus remains constant for all configurations. Using row based fabrics makes hardware design and application mapping easier and significantly reduces the routing complexity over traditional FPGA architectures. It also makes partial reconfiguration of the fabric easier as designs occupy a certain number of rows; so long as that number of rows is available in the fabric, it can be reconfigured by reprogramming only those rows.

Virtualization, such as that employed by PipeRench [13, 14], allows the fabric to handle configurations that require more rows than are physically available in the SPL. The

costs of virtualization are degraded throughput, since the design can no longer be fully pipelined, and higher power. Despite these drawbacks, virtualization is a key component of efficient spatial sharing.

3. SPL ARCHITECTURE

3.1. Motivating Example

We motivate our work by demonstrating the drawbacks of a straightforward application of prior SPL approaches to a CMP, and the substantial benefits of intelligent SPL sharing. Figure 1(a) shows a possible floorplan – with areas drawn to scale – for a multicore with eight single-issue out-of-order cores, each of which is coupled with a row-based fabric (L2 cache not shown). Each fabric contains 26 rows of programmable logic, just enough to avoid virtualization for all eight applications. (Application statistics and modeling methodology are described later.)

The SPL area is roughly twice that of each core. Granted, a more complex core would consume more area, and we explore such options later in the paper. Still, from an area perspective, there is clearly room for improvement. Figure 2 shows the utilization (percentage of the total number of rows that are in use on average) and performance of several SPL configurations for a coarse-grain multithreaded workload of eight single-threaded applications, each of which runs on one of the eight cores. The leftmost bars for the individual benchmarks show the utilization of the 26-row configuration of Figure 1(a). The utilization of seven of the SPL fabrics is less than 10% (with the eighth at about 30% utilization), and the average SPL utilization is only 7%. Reducing each SPL to 12 rows (next set of bars) markedly increases SPL utilization for some benchmarks; moreover, the area is greatly reduced as shown in Figure 1(b). However, this comes at a high cost: an 18% overall performance loss, since all benchmarks use more than 12 rows. Moving to six-row fabrics further improves utilization but at a 49% performance penalty.

By intelligently sharing the SPL among multiple cores, the average number of rows for each core *can* be reduced to six with little performance loss relative to the private 26-row configuration. Figure 1(c) shows a floorplan with two pools of SPL, each of which contains 24 rows and is shared – using control policies that we describe in Section 3.5 – among four cores. This configuration reduces the SPL area and peak power costs by over 4X. Furthermore, as shown by the fourth AvgUtilization bar[1] in Figure 2, the average utilization of the fabrics increases threefold. These benefits come with virtually the same performance as the 26-row private configuration. The rightmost bars show that a single 48-row SPL shared among all eight cores further improves utilization, and also suffers negligible performance loss.

The contrast in performance between the Private six-row and four-way shared SPL configurations – which have the same total number of SPL rows – motivates the need for good sharing policies. The six-row Private configuration can be viewed as a spatially shared SPL organization with a naïve control policy that equally divides the SPL among all cores at all times. The shared configurations use a more intelligent policy that eliminates the 49% performance loss of the naïve approach.

[1]We do not show utilization for individual benchmarks since the fabrics are shared among multiple benchmarks.

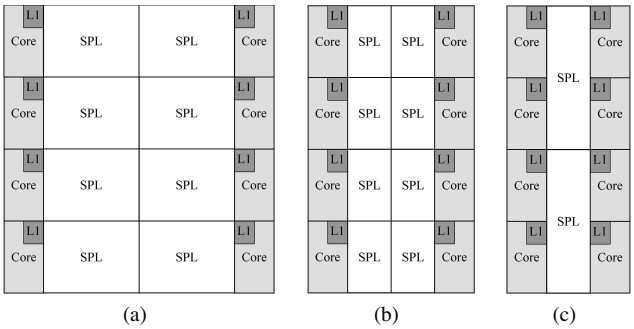


Fig. 1: Sample floor plans for an eight-way multicore with (a) 26-row private, (b) 12-row private, and (c) 4-way shared SPL.

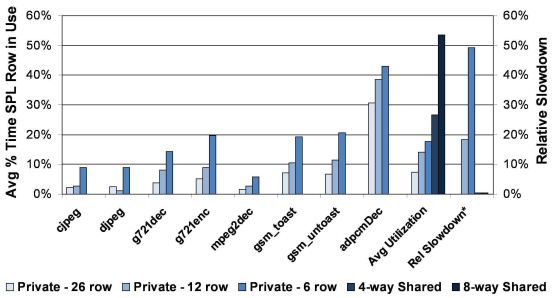


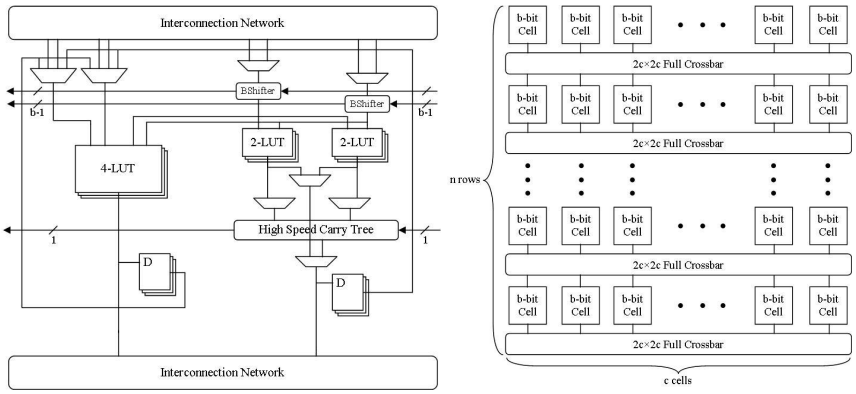
Fig. 2: SPL utilization for private and shared SPL organizations. *The last set of bars shows percentage slowdown relative to 26-row private SPL.

3.2. Fabric Microarchitecture

Each SPL fabric consists of n rows, in which each row contains c cells, and each cell computes b bits of data. Figure 3(a) shows the cell design. The major components are a main 4-input look-up table (4-LUT), a set of two 2-LUTs (equivalent to one 3-LUT) plus a fast carry chain to compute carry bits or other logic functions if carry calculation is not needed, barrel shifters to properly align data as necessary, flip-flops to store results of computations, and an interconnect network between each row. Within a cell, the same operation is performed on all b bits. These b-bit cells are arranged in a row to form a c×b-bit row. Each cell in a row can perform a different operation on its inputs and a number of these rows are grouped together to execute an application function. Each row completes in a single SPL clock cycle.

As shown previously [14], several tradeoffs dictate the optimal choice of bit width, row width, and number of rows. Increasing the bit width decreases area and power at the cost of less configuration flexibility. Increasing the row width allows more computation in a single cycle but also increases the likelihood of less than 100% resource utilization if not all of the cells in a row can be put to use. Furthermore, the fabric width should match the ability of the memory system to supply data at a fast enough rate. Finally, reducing the number of rows has linear area and power benefits but increases the number of functions that must be virtualized.

To quantitatively evaluate these tradeoffs, we created analytical area, latency, and power models for SPL in 65nm technology. The model is created by combining estimates for the different components of the fabric. We use Cacti 4.2 [17] to model LUTs, Orion [18] for between row interconnect modeling, the models of Heo and Asanovic [19] for local wiring and between cell wires (such as for the barrel shifter and carry logic not included in the Orion model), and



(a) (b)

Fig. 3: (a) SPL cell design and (b) integration in overall fabric. Unless indicated, all data paths are b bits wide.

Table 1: Comparison of data from actual reconfigurable fabrics (scaled to 65nm) and the analytical model.

	Scaled Actual	Model	% Diff
PipeRench - 8-bit, 16-cell, 16-row - 180 nm			
Area (mm^2)	1.5	1.59	6.0%
Frequency (MHz)	350	460	31.4%
Power (W)	0.929	0.832	-10.5%
Chimaera - 1-bit, 32-cell, 32-row - 0.6 μm			
Area (mm^2)	0.805	0.751	-6.7%
Frequency (GHz)	1.27	1.06	-16.5%
Garp - 2-bit, 24-cell, 32-row - 0.5 μm			
Area (mm^2)	1.32	1.06	-19.7%

the work of [20] for the carry chain logic implementation. Finally, various bit level components, such as transistor delay, area, and power, used to compute estimates for muxes and small SRAM cells, are taken from the ITRS [21].

To validate our model, we compare scaled values of area, latency, and power available from previous reconfigurable fabric designs [12, 15, 22] with predictions by our model for these fabric architectures. The results are given in Table 1. The Scaled Actual Area values in this table are derived by scaling the fabric area of each design by the square of the ratio of the technology factors. For frequency, the reported values are linearly scaled by the technology ratio. The Scaled Actual Power value for PipeRench is derived by scaling the reported power value by the square of the ratio of the PipeRench voltage to the SPL voltage, by the ratio of the PipeRench frequency to the SPL frequency, and by the ratio of the technology factors (to account for capacitance scaling). The model achieves good correlation except for the frequency in PipeRench and Chimaera and the area of Garp. For PipeRench, the PipeRench paper notes that the circuit design was not highly optimized [15], and therefore we expect that a frequency closer to our higher value could be achieved in an industrial PipeRench design. Given the disparity in technologies between our design and Chimaera, a 16% error is not unexpected. For the area disparity with Garp, the information available is limited, making good correlation with the Garp design difficult.

Figure 4 shows area and latency comparisons between our model and [23] in which area and latency values were derived for different numbers of row inputs and outputs. The model error is within 15% in all but two cases.

We use the model to estimate the costs of different configurations for the various functions that we map to the SPL (discussed in Section 4). An 8-bit wide cell with 128-bit

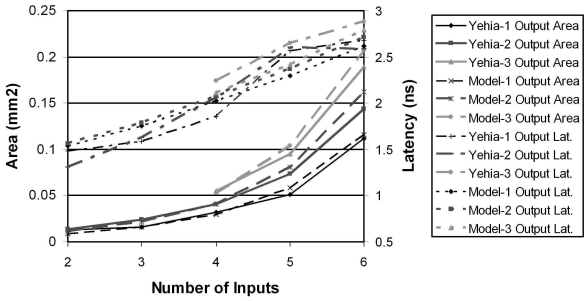


Fig. 4: Comparison of latency and area predicted by model to results from Yehia et al. [23].

Table 2: Area and power of different core types and 26-row SPL normalized to IO area and power.

	Area	Dyn. Power	Leakage Power
IO	1.00	1.00	1.00
OOO1	1.19	1.06	1.05
OOO2	1.82	1.26	1.26
OOO4	4.87	1.66	1.63
SPL	2.47	0.66	3.00

wide rows provides a good compromise between flexibility and area/power cost, and permits significant parallelization. Each SPL function can take in up to 512 bits of input and can produce up to 128 bits of output. This organization achieves a reasonably high frequency (500MHz) relative to the processor core frequency (assumed 2GHz at 65nm). At this 1/4 clock speed differential, four quadword load instructions can supply 512 bits to the SPL pipeline every SPL clock period. Finally, for the baseline 26-row private SPL, we set the number of configurations to eight, the maximum needed during any program phase for our workloads.

Table 2 shows the area and power of a 26-row SPL compared with four conventional core types: an in-order core (designated as IO), and one-, two-, and four-way out-of-order cores (OOO1, OOO2, and OOO4). Results are normalized relative to the IO core. Each core has separate 8KB L1 instruction and data caches. We adopted the methodology of Kumar et al. [24] to calculate per-core area and power costs. We note that an OOO1 core augmented with SPL has an area that falls between OOO2 and OOO4, while the area of OOO2 + SPL is slightly less than that of OOO4.

We create shared SPL configurations by pairing OOO1 cores with SPL that consumes approximately half the area of the cores. A six row SPL consumes slightly more than half

Table 3: SPL configurations and area and power costs.

	Rows/ SPL	Configs/ Row	Total SPL Area (mm^2)	Dynamic Energy/ Row (nJ)	Total SPL Leakage Power (W)
Eight Private	26	8	23.52	.0600	4.58
Four 2-way Shared	12	8	5.45	.0601	1.06
Two 4-way Shared	24	10	5.83	.0601	1.07
One 8-way Shared	48	12	6.23	.0601	1.09

the area of an OOO1 core and the combination of the two is smaller than OOO2. Given these constraints, we arrive at the shared SPL configurations shown in Table 3, which also includes the baseline private 26-row SPL organization for comparison purposes. In terms of total area and leakage power, four two-way shared SPLs come at slightly less than a quarter of the cost of the eight private SPLs. The reduced number of rows, however, means that functions will need to be virtualized more often. Indeed, any function requiring all 26 rows of the private configuration will be virtualized in both a two- and four-way shared SPL, even if the other cores are not using the shared SPL at that time. SPL configurations with a higher degree of sharing consume a slightly larger area than the two-way shared configuration but they require less virtualization. When several applications simultaneously reach a point where they do not need the SPL, it can be allocated among the remaining cores so that virtualization can be avoided. The area increase for higher degrees of sharing comes from the additional datapath hardware and on-chip configurations for higher degrees of sharing, with the latter contributing most of the increase.

3.3. Core/Fabric Integration and Shared Fabrics

As with a number of previous designs [4, 5, 6, 7], the SPL fabric is tightly integrated with the processor core as a reconfigurable functional unit. However, rather than consume additional register file ports, we use a queue-based decoupled architecture to interface the SPL to the memory system. The queue matches the SPL row input width (512 bits) and special SPL load instructions place values into the queue at a particular data alignment. The queue accepts input data at the 2GHz core frequency such that four quadword loads match the maximum input bandwidth of the fabric. Likewise, instead of writing to the register file, the SPL writes to a single output staging register that is then written out to the Store Queue using a special SPL store instruction. Since the normal LSQ/cache datapath is used for data transfer, no additional steps are needed to handle memory dependences with the processor core.

As discussed in Section 2, row-based designs employing virtualization are highly amenable to sharing. Figure 5(a) shows how our row-based SPL is modified to enable spatial sharing between two cores. Additional muxes select input bits at the entry point of the SPL and at each point where the SPL pool might be partitioned. Furthermore, an additional set of tristate drivers tap off of each row output to drive the sharer's output register. Finally, there is additional wire overhead to get data to and from multiple cores. These wires can be pipelined if necessary to match the SPL clock frequency at the cost of additional pipeline initiation time. However, with deeply pipelined row-based fabrics, the cost is small and is outweighed by the efficiency gains.

For temporal sharing (see Figure 5(b)), all rows of the fabric are available to all cores in a time multiplexed fashion. Therefore, no intermediate muxes are required.

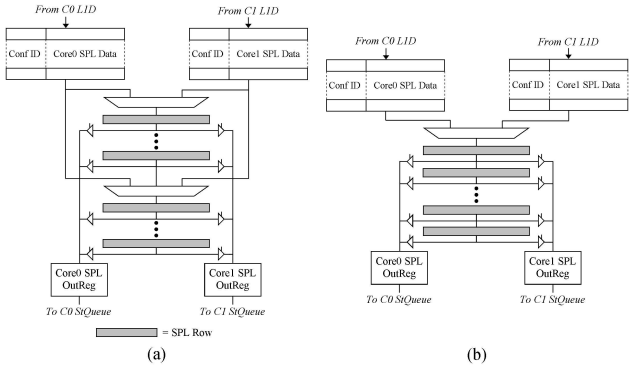


Fig. 5: Two core (a) spatial sharing and (b) temporal sharing.

Table 4: ISA extensions.

Instruction	Description
`spl_lsize align, offset(reg)`	load data of size *size* into SPL input queue at alignment *align*
`spl_ssize align, offset(reg)`	store data of size *size* from SPL output register to memory at alignment *align*
`spl_initiate config`	Invoke SPL using configuration *config*
`spl_prefetch config`	Prefetch SPL configuration *config* into configuration memory

3.4. Software Interface

To expose the SPL to software, we extend the ISA with the instructions shown in Table 4. These instructions handle moving data to and from the SPL and initiating the execution of a SPL instruction. In an actual system these instructions could be generated by a compiler that supports reconfigurable architectures such as those proposed in [11, 13, 25].

3.5. Sharing Policies

We explore hardware-level control policies for spatial and temporal SPL sharing. The advantage of spatial sharing is that each core has dedicated resources and is guaranteed to make forward progress each cycle, although possibly at a degraded rate due to increased virtualization. Temporal sharing is almost diametrically opposed to spatial sharing. While all cores must vie for the same SPL resources, those resources are large enough that virtualization is likely to occur less often, perhaps even less so than with private SPLs. We discuss each of these approaches in turn.

3.5.1. Spatial Sharing Algorithm

The first decision with spatial sharing is how finely to divide up the fabric. One can choose an equal number of rows based on the number of sharers, i.e., the SPL is split into n equal sections if there are n sharers, or split according to expected application usage. These approaches require a large number of intermediate muxes. We investigated a simpler alternative that splits by only powers of two; if, for example, there are 5-8 sharers, the SPL will be broken into eight partitions and some of these may lie unused.

To determine when to merge SPL partitions, per-core idle cycle counters and an idle count threshold value (1000 in our implementation) are associated with each shared SPL pool. When a core has no SPL instructions in-flight, its idle counter is reset. The counter counts up each cycle that the core does not request use of the SPL. Once the threshold is reached, the SPL checks to see if the number of threads now using the SPL falls within a different power of two partition. If so, it waits for all current in-flight functions to finish and

Table 5: Benchmark, number of SPL functions, maximum rows used by SPL functions, percentage of execution time replaced by SPL functions, percentage of SPL instructions executed relative to total committed instructions, and percentage of time with at least one SPL instruction in flight, for OOO1 cores.

	SPL Functions	Max Rows	% Replaced Exec Time	% Dyn. SPL Insts	SPL Usage
cjpeg	5	21	49.9%	1.19%	17.77%
djpeg	3	23	61.9%	0.84%	9.72%
g721dec	1	26	48.1%	0.71%	27.94%
g721enc	1	26	45.5%	0.67%	24.11%
mpeg2dec	3	10	62.9%	1.07%	22.28%
gsm_toast	2	16	54.2%	2.83%	28.92%
gsm_untoast	1	22	75.8%	2.18%	36.55%
adpcmDec	1	24	95.9%	10.29%	79.22%
MPGenc	4	16	69.1%	0.72%	17.23%
MPGdec_o0	5	20	44.8%	0.35%	15.30%
MPGdec_o3	12	20	47.8%	0.57%	19.25%
crypt	1	298	97.9%	4.48%	99.90%

then repartitions the SPL, allocating the core's partition to the other active sharers.

3.5.2. Temporal Sharing Algorithm

For temporal sharing, we share the fabric among the cores in a round-robin manner on an SPL cycle-by-cycle basis. The per-core queues permit each core to continue to issue SPL instructions while it awaits its turn in the rotation.

4. EVALUATION METHODOLOGY

Our simulator is a highly modified version of SESC [26]. We assume processors implemented in 65 nm technology running at 2.0 GHz with a 1.1V supply voltage. We use Wattch and Cacti to model dynamic power and Cacti and HotLeakage to model leakage power.

Since the SPL versions of each benchmark execute a significantly different number of instructions than the baseline, we compare execution times for complete program execution as IPC comparisons are meaningless. To make our comparisons more accurate, we continuously respawn jobs that finish before the longest running thread in order to maintain a consistent SPL access pattern.

4.1. Benchmarks

For our coarse grain workload, we use five benchmarks from the MediaBench suite [27] and three benchmarks from the MediaBench II suite [28]. For parallel workloads, we select two benchmarks from the ALPBench suite [29] and a version of the Java Grande [30] crypt benchmark ported to C++. We run the ALPBench version of MPGdec with two different command line parameters (-o0 and -o3) as they produce drastically different execution characteristics.

Benchmark statistics are given in Table 5. All but three of the benchmarks use at least 20 rows, but few of these occupy the SPL for any great length of time. This indicates that for our workloads and architecture, there may be more opportunity for temporal rather than spatial sharing.

5. RESULTS

Prior work has shown that SPL coupled with a standard processor can significantly speed up certain application classes, e.g., multimedia workloads. We achieve similar results as shown in Figure 6, which plots the performance improvement of adding 26-row private SPL to each core. For each benchmark, the values are normalized to the performance

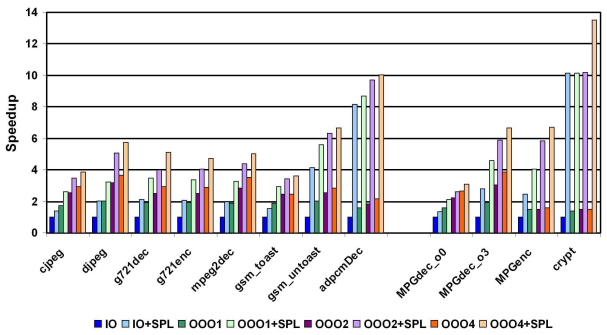


Fig. 6: Performance relative to IO cores without SPL.

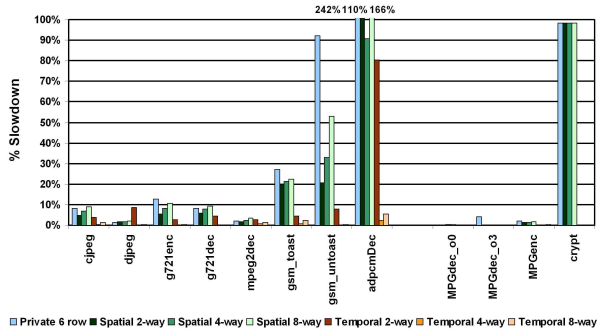


Fig. 7: Performance relative to OOO1 + 26 row private SPL.

on an IO core without SPL. In all eight coarse grain benchmarks and three of the parallel benchmarks, a CMP with an n-way OOO core plus SPL outperforms the next larger OOO core; in eight of these cases the OOO1+SPL outperforms the OOO4 core, which consumes far more area and power.

We quantify the shared versus private SPL performance for a CMP of OOO1 cores. We compare the shared configurations to two versions of the private SPL: one with the full 26 rows, and another with six rows, the same per core amount as the shared cases (Table 3).

With two- and four-way sharing, we need to consider which applications should be scheduled together on the same SPL pool. Our algorithm statically schedules threads based on the percentage of time they use the SPL, with the objective of roughly equalizing SPL usage. We schedule *adpcmDec*, the highest utilization thread with *djpeg*, the lowest utilization one (see Table 5 for usage values). We then pair the next highest with the next lowest on the next SPL pool, and so on until all threads are scheduled.

The performance and energy×delay results for all benchmarks are shown in Figures 7 and 8, respectively. Sharing SPL pools reduces energy×delay by up to 33% overall compared to the use of private SPL, with little or no performance degradation with the proper sharing policy. The SPL area is also reduced by more than 4X.

The private SPL configuration with only six rows is not an acceptable alternative to fabric sharing. For three of the benchmarks – *gsm_untoast*, *adpcmDec*, and *crypt* – the performance and energy×delay costs are significant relative to the 26-row private SPL configuration. Spatial sharing improves slightly upon the 6-row private SPL organization for two of these three cases, but has the least benefit for *crypt*. This workload experiences almost a 100% performance slowdown – and a larger energy×delay penalty – with spatial

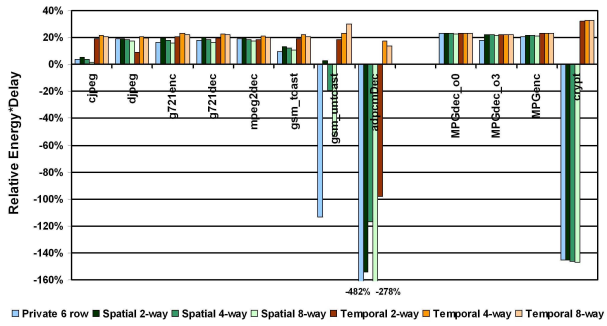


Fig. 8: Energy×Delay relative to OOO1 + 26 row private SPL.

sharing relative to 26-row private SPLs.

Temporal sharing, especially four- and eight-way sharing, outperforms spatial sharing overall for two primary reasons. First, all benchmarks but crypt need a maximum of 26 rows for all functions; thus, with four- and eight-way temporal sharing virtualization is rarely required. Second, there are significant periods where the benchmarks access the SPL at a slow enough rate to intersperse requests from different cores. With eight-way sharing, however, the performance of several benchmarks degrades non-trivially due to a 40% increase in input queue wait time due to increased SPL conflicts. In any event, the temporal four-way sharing approach achieves neglible performance loss, and significant energy savings, for every benchmark relative to the private SPL organization at far less area and peak power costs.

6. CONCLUSIONS

In this paper, we propose a reconfigurable architecture that uses a shared fabric and control policies to reduce the costs of marrying reconfigurable logic and processor cores in future CMPs. Using intelligent sharing policies, our approach requires 4X less area and peak power than private SPL while achieving the same performance improvement over area-equivalent conventional CMPs.

7. REFERENCES

[1] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *Computer–Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 203–215, 2007.

[2] R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective," in *Proc. of the Conference on Design, Automation, and Test in Europe*, 2001, pp. 642–649.

[3] T. Todman, G. Constantinides, S. Wilton, P. Cheung, W. Luk, and O. Mencer, "Reconfigurable Computing: Architectures and Design Methods," *IEE Proc. – Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–205, March 2005.

[4] R. Razdan and M. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," in *Proc. IEEE/ACM 27th International Symposium on Microarchitecture*, 1994, pp. 172–180.

[5] M. Dales, "Managing a Reconfigurable Processor in a General Purpose Workstation Environment," in *Proc. of the Design, Automation, and Test in Europe Converence and Exhibition*, 2003, pp. 980–985.

[6] Stretch Inc., *The S6000 Family of Processors*.

[7] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," in *Proc. 27th IEEE/ACM International Symposium on Computer Architecture*, 2000, pp. 225–235.

[8] A. DeHon, "DPGA Utilization and Application," in *Proc. 1996 ACM/SIGDA 4th International Symposium on Field Programmable Gate Arrays*, 1996, pp. 115–121.

[9] M. Wirthlin and B. Hutchins, "A Dynamic Instruction Set Computer," in *Proc. 1995 IEEE Symposium on Field–Programmable Custom Computing Machines*, 1995, pp. 99–107.

[10] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, and H. Holt, "The NAPA Adaptive Processing Architecture," in *Proc. 1998 IEEE Sym. on Field–Programmable Custom Computing Machines*, 1998, p. 28.

[11] T. Callahan, J. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler," *Computer*, vol. 33, pp. 62–69, Apr. 2000.

[12] J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proc. 1996 IEEE Symposium on Field–Programmable Custom Computing Machines*, 1997, pp. 12–21.

[13] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *Computer*, vol. 33, pp. 70–77, 2000.

[14] S. Goldstein, H. Schmit, M. Moe, M. Budiu, and S. Cadambi, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," in *Proc. 26th IEEE/ACM International Symposium on Computer Architecture*, 1999, pp. 28–39.

[15] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Taylor, "PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology," in *Proc. of the IEEE 2002 Custom Integrated Circuits Conference*, 2002, pp. 185–192.

[16] J. Carrillo and P. Chow, "The Effect of Reconfigurable Units in Superscalar Processors," in *Proc. 2001 ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays*, 2001, pp. 141–150.

[17] D. Tarjan, S. Thoziyoor, and N. Jouppi, "CACTI 4.0," *HP Technical Report*, 2006.

[18] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik, "Orion: A Power-Performance Simulator for Interconnection Networks," in *Proc. IEEE/ACM 35th International Symposium on Microarchitecture*, 2002, pp. 294–305.

[19] S. Heo and K. Asanovic, "Replacing Global Wires with an On-Chip Network: A Power Analysis," in *ISLPED '05: Proc. of the 2005 International Symposium on Low Power Electronics and Design*, 2005, pp. 369–374.

[20] S. Hauck, M. Hosler, and T. Fry, "High-Performance Carry Chains for FPGA's," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, pp. 138–147, 2000.

[21] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors," 2007.

[22] S. Hauck, T. W. Fry, M. M. Holser, and J. P. Kao, "The Chimaera Reconfigurable Functional Unit," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 207–217, Feb. 2004.

[23] S. Yehia, N. Clark, S. Mahlke, and K. Flautner, "Exploring the Design Space of LUT-based Transparent Accelerators," in *CASES '05: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2005, pp. 11–21.

[24] R. Kumar, D. Tullsen, and N. Jouppi, "Core Architecture Optimization for Heterogeneous Chip Multiprocessors," in *Proc. IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, 2006, pp. 23–32.

[25] M. Budiu and S. C. Goldstein, "Fast Compilation for Pipelined Reconfigurable Fabrics," in *Proc. 1999 ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 1999, pp. 195–205.

[26] "SESC Architectural Simulator," `http://sourceforge.net/projects/sesc`, 2007.

[27] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A Tool for Evaluation and Synthesizing Multimedia and Communications Systems," in *Proc. IEEE/ACM 30th International Symposium on Microarchitecture*, 1997, pp. 330–335.

[28] J. Fritts, F. Steiling, and J. Tucek, "MediaBench II Video: Expediting the Next Generation of Video Systems Research," in *Proc. of The Int'l Society for Optical Engineering*, vol. 5683, 2005, pp. 79–93.

[29] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Deves, "The ALPBench Benchmark Suite for Complex Multimedia Applications," in *Proc. of the IEEE International Symposium on Workload Characterization*, 2005, pp. 34–45.

[30] L. Smith, J. Bull, and J. Obdrzálek, "A Parallel Java Grande Benchmark Suite," in *Supercomputing '01: Proc. of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, 2001, pp. 8–17.