Dynamically Managed Multithreaded Reconfigurable Architectures for Chip Multiprocessors

Matthew A. Watkins
Computer Systems Laboratory
Cornell University, Ithaca, NY
maw72@cornell.edu

David H. Albonesi
Computer Systems Laboratory
Cornell University, Ithaca, NY
albonesi@csl.cornell.edu

ABSTRACT

Prior work has demonstrated that reconfigurable logic can significantly benefit certain applications. However, reconfigurable architectures have traditionally suffered from high area overhead and limited application coverage. We present a dynamically managed multithreaded reconfigurable architecture consisting of multiple clusters of shared reconfigurable fabrics that greatly reduces the area overhead of reconfigurability while still offering the same power efficiency and performance benefits. Like other shared SMT and CMP resources, the dynamic partitioning of the reconfigurable resource among sharing threads, along with the co-scheduling of threads among different reconfigurable clusters, must be intelligently managed for the full benefits of the shared fabrics to be realized.

We propose a number of sophisticated dynamic management approaches, including the application of machine learning, multithreaded phase-based management, and stability detection. Overall, we show that, with our dynamic management policies, multithreaded reconfigurable fabrics can achieve better energy$\times$delay^2, at far less area and power, than providing each core with a much larger private fabric. Moreover, our approach achieves dramatically higher performance and energy-efficiency for particular workloads compared to what can be ideally achieved by allocating the fabric area to additional cores.

Categories and Subject Descriptors

C.5.3 [**Computer System Implementation**]: Microcomputers—*Microprocessors*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable Architectures*

General Terms

Design, Performance

Keywords

Shared Resource Management, Reconfigurable Architecture

PACT'10, September 11–15, 2010, Vienna, Austria.

1. INTRODUCTION

Reconfigurable logic has been proposed as one possible way to improve the performance and power efficiency of microprocessors [23, 41]. Researchers have proposed specialized fabrics that are specifically designed for more efficient integration with general purpose processors than conventional FPGAs [4, 22, 44]. Despite these advances in fabric architecture, reconfigurable logic still incurs non-trivial power and area costs relative to the fixed hardware functionality of commercial microprocessors [28]. These costs are especially important given the disparity in benefit that different applications can expect to receive from reconfigurable fabrics, from orders of magnitude benefit to no benefit at all.

The ability to integrate multiple cores and reconfigurable logic on a single die afforded by the billion transistor era provides a new opportunity to address these issues. This paper proposes dynamically managed multithreaded reconfigurable fabrics that, similar to shared SMT resources and last level caches, are shared among several cores of a CMP in order to save area and increase fabric utilization. Since the degree of fabric sharing must necessarily be limited, multiple clusters of cores sharing a common *Specialized Programmable Logic (SPL)* fabric may be implemented, depending on the expected percentage of applications that can be accelerated by the SPL. Like other shared resources in a CMP of SMT cores, where the partitioning of resources among the competing threads on a given SMT core and the co-scheduling of threads to multiple SMT cores significantly impact performance, the control of multiple multithreaded SPL clusters must be intelligently managed for good performance to be achieved. Specifically, such a manager must make two interrelated decisions: (1) determine the best match of threads to the multiple clusters of SPL, considering the interplay between different threads; and (2) decide when and how best to spatially partition each fabric on-the-fly in order to reduce contention among the threads, at the potential cost of degraded throughput.

In this paper, we explore a number of approaches to this complex management problem that range in sophistication from simple interval-based heuristic approaches to more advanced techniques that apply machine learning, multithreaded phase optimization, and stability analysis. Our algorithms permit the use of very compact SPL fabrics that are performance competitive (on multiple mixed sequential and parallel workloads with high SPL demand) with large private SPL attached to each core, while consuming several times less die area and energy. Moreover, we show that replacing the SPL with additional cores degrades performance

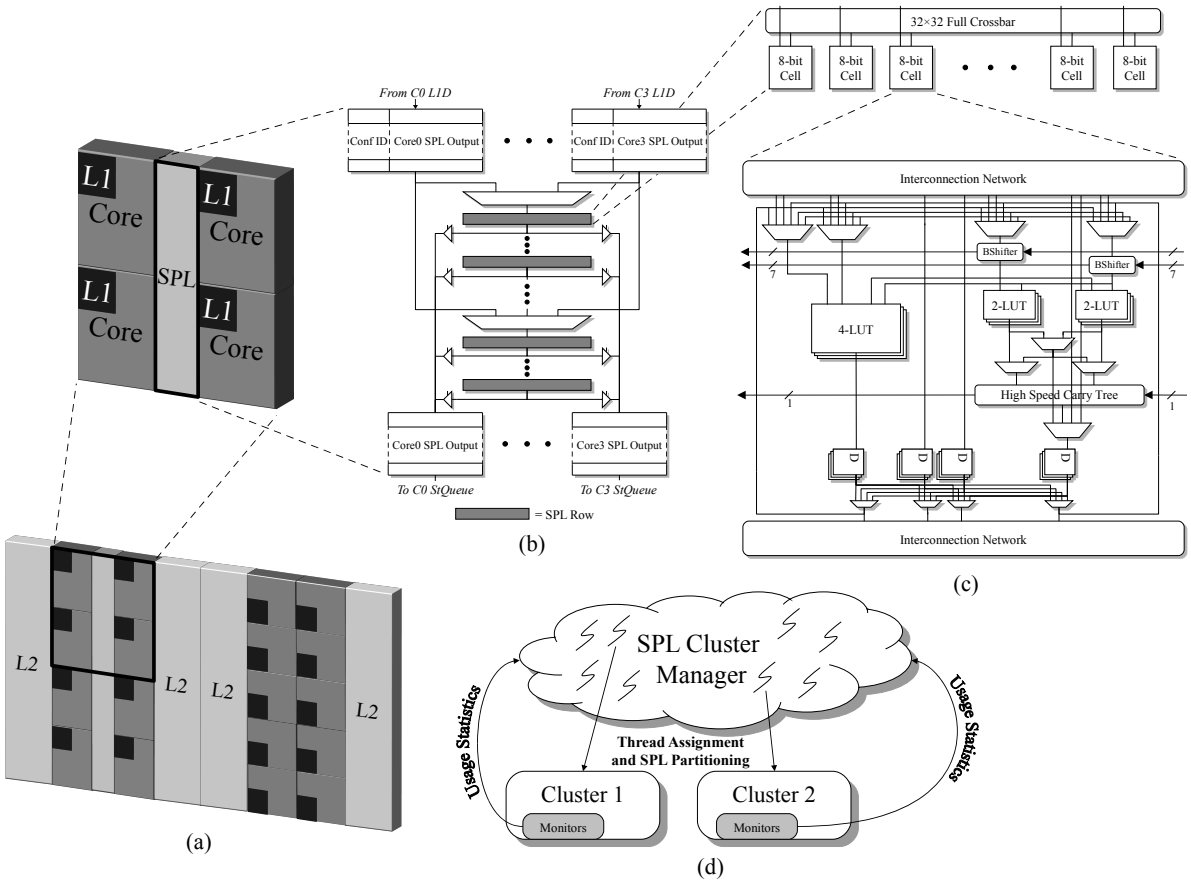


Figure 1: Overview of SPL integration in a CMP. (a) Depiction of overall chip, with two SPL clusters and one conventional cluster, and blow-up of one SPL cluster, (b) four-way multithreaded SPL, (c) design of SPL cell (unless otherwise indicated all data paths in SPL are 8 bits wide), and (d) SPL Cluster Manager.

by 62-143% for our workloads, demonstrating the benefit of dynamically managed clusters of multithreaded SPL.

In the next section, we describe the architecture of a CMP with embedded SPL, followed by our management policies in Section 3. Our evaluation methodology is presented in Section 4. Section 5 evaluates our approach. We describe related work in Section 6 and conclude in Section 7.

2. MULTITHREADED RECONFIGURABLE ARCHITECTURES

Figure 1(a) shows an overall depiction of a hypothetical 18 core CMP with three clusters[1], with the external interface not shown for simplicity. Each of the two clusters on the left hand side consists of a multithreaded SPL fabric shared by four single issue out-of-order processor cores. In our previous work [43], we evaluated the use of SPL with a range of in-order and out-of-order core types and found that a simple out-of-order core coupled with SPL provided the best area-equivalent performance and power efficiency. Moreover, similar to SMT processors where adding additional contexts provides limited benefit beyond a certain point [42], sharing an SPL among four cores was shown to be the best trade-off between SPL fabric utilization and contention among competing threads. To confirm that this result holds true for our set of workloads, we evaluated systems with both two 4-way and one 8-way shared SPL and found that, in all cases, two 4-way shared SPL clusters outperformed a single 8-way shared SPL. While the techniques presented in this paper apply to any degree of sharing, we assume a 4-way shared SPL in the remainder of this paper.

In the "conventional" cluster on the right hand side of Figure 1(a), each SPL has been replaced by one additional core, giving 10 cores in total. Applications that do not benefit from the SPL run on this conventional cluster, while those that can exploit the SPL run on one of the two left clusters. Of course, different mixes of SPL and conventional clusters (as well as other cluster types) are possible, but this consideration is beyond the scope of this paper.

Each SPL, adopted from [43] and shown in more detail in Figures 1(b) and (c), is a highly pipelined row-based [22, 44] programmable fabric that is temporally shared among the four cores. Each of the two clusters incorporates hardware monitors that capture cycle-level event counts relevant to application characteristics and SPL usage. As is shown in Figure 1(d), the SPL Cluster Manager periodically reads the monitored information in order to assign threads to clusters, and to spatially partition each SPL as appropriate to optimize performance and power efficiency. In the remainder of Section 2, we provide an overview of the SPL microarchitecture. A more complete treatment is available in [43].

[1]Although relative sizes of the cores and SPL are accurate, this is not intended to represent an actual floorplan.

	Rows/ SPL	Total Area	Peak Dynamic Power	Total Leakage Power
Eight Cores	N/A	1.00	1.00	1.00
Eight Private SPL	12	0.97	0.29	1.32
Two 4-way Shared SPL	12	0.29	0.07	0.34

Table 1: Relative area and power of eight single-issue out-of-order cores, eight private SPLs, and two four-way shared SPLs.

2.1 SPL Hardware Microarchitecture

We adopt the row-based fabric of our earlier work [43] in which the space of shared SPL configurations was explored using validated SPL delay, power, and area models. The SPL is tightly integrated with the processor core as a reconfigurable functional unit, and is interfaced to the memory system via a queue-based decoupled architecture (Figure 1(b)). The input queue matches the SPL row input width (512 bits) and special SPL load instructions place values into the queue at a particular data alignment. Likewise, the SPL writes to an output queue and the head entry is written out to the Store Queue using special SPL store instructions. Since the normal LSQ/cache datapath is used for data transfer, no additional steps are needed to handle memory dependencies with the processor core.

The SPL itself is composed of 8-bit wide computation cells. The same operation is performed on all 8 bits within a cell. Sixteen of these 8-bit cells are arranged in a row to form a 128-bit wide row. Each cell in a row can perform a different operation on its set of inputs and n of these rows are grouped together to form the overall SPL fabric. Figure 1(c) shows the row and cell design. Feedback within a single row is allowed. For our benchmarks, 12 rows of private (per-core) SPL permits all but one of the configurations, the major loop within *crypt*, to achieve maximum performance[2]. In 65 nm technology, the SPL can be clocked at 500 MHz, one-fourth that of the processor core frequency of 2GHz (the same as the Pentium Core2 Duo [24] and the AMD X2 Dual-Core [1], both of which are implemented in 65 nm). This latency permits each row to complete the longest possible computation in a single cycle. The SPL includes integrated on-chip storage for 12 configurations to allow for fast switching between different configurations. For our workloads, this permits all configurations for any phase to reside on-chip. Thus, reconfiguration latency is not an issue as all configurations are immediately available after the initial configuration overhead is paid.

Using our analytical models [43], we arrive at the area and power results for eight single issue out-of-order cores, eight private 12-row SPLs, and two four-way shared SPLs shown in Table 1. Although the area is prohibitive, the 12-row private SPL serves as the baseline for comparison with the dynamically managed multithreaded SPL architecture. The latter is much more compact, requiring 4X less area than the private SPLs, and is much more power-efficient as well. While one might consider shrinking the private SPL even further, our previous work [43] has shown that that this yields poor performance.

[2]The major loop in *crypt* requires nearly 300 rows and so achieves less than optimal speedup for any reasonably sized fabric.

2.1.1 SPL Virtualization

Virtualizing reconfigurable hardware was proposed by [3] to allow a fabric to execute a configuration that requires more resources (i.e., rows) than are physically available. Although throughput is reduced when the design must be virtualized, virtualization permits the designer to trade performance for area. As more area becomes available (or for higher-end chips) larger fabrics can be created without requiring any change to the application mappings.

Virtualization is accomplished by using the same physical row to execute multiple virtual rows. For example, when executing a configuration requiring six rows on a fabric with only three physical rows, virtual rows 1 and 4 execute in physical row 1, virtual rows 2 and 5 in physical row 2, etc. In this example, this leads to a maximum 50% reduction in throughput relative to the unvirtualized case as new data can be inserted only half as often.

For shared fabrics the number of rows available to a function is not known at application design time even for a particular fabric implementation as the function may not be allocated the entire SPL. As such, virtualization is especially useful for shared fabrics as it allows all SPL functions to be executed, albeit with possibly different throughput, regardless of the number of rows that are allocated at runtime.

2.1.2 Temporal Sharing and Spatial Partitioning

Figure 1(b) shows how the SPL design permits temporal sharing among the threads executing on the four processor cores, as well as spatial partitioning to permit private or semi-private operation [43]. Spatial partitioning is enabled by inserting additional multiplexers at each point where the SPL pool might be partitioned. To keep the hardware overhead reasonable, each of the two SPLs can only be divided into two halves, four quarters, or one half and two quarters, requiring a total of only four sets of input multiplexers.

With temporal sharing, all rows of the temporally shared fabric (which may be the whole SPL or a subset, depending on whether spatial partitioning is being used) are available to the sharing threads in a time multiplexed fashion. The control for temporal sharing is implemented in hardware on a fine-grain, cycle-level basis to avoid thread starvation for the shared SPL resource. Each SPL cycle, a round-robin scheduler selects an instruction from one of the queues to issue to the shared SPL.

2.2 SPL Function Mapping

The SPL is used to accelerate a wide range of operations. We show one such example from the SPEC 2006 application *456.hmmer*. We accelerate the P7Viterbi function, which accounts for 85% of the program execution time. The core loop of the function is shown in Figure 2(a). Figure 2(b) shows how the portion of the code that calculates `mc` is mapped to the SPL. In the optimized code, the core first loads the input values needed to compute `mc` into the fabric, the SPL computes the value of `mc`, and finally the core receives the result. After receiving `mc`, the core computes the values of `dc` and `ic` and repeats the loop.

3. DYNAMIC MANAGEMENT OF MULTI-THREADED FABRICS

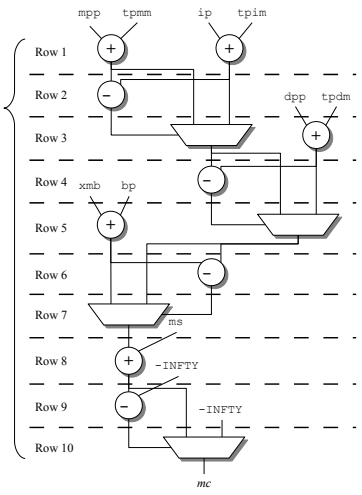
Having provided background on the fabric architecture, we now present our dynamic runtime management policies

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1] + tpmm[k-1];
  if ((sc = ip[k-1] + tpim[k-1]) > mc[k]) mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
  if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;

  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;

  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

(a) Source code



(b) Calculation of mc in SPL

Figure 2: Mapping of SPEC2006 *456.hmmer* P7Viterbi to SPL.

Consideration	Alternatives
Metrics	SPL Accesses, SPL Wait Time, Avg. Rows, Grad. Insts.
Spatial Partitioning	Yes, No
Granularity	Various fixed intervals, phase change
Algorithm	Split Assignment, Equalize Assignment, Hill Climbing, Hybrid
Stability	None, after n non-useful changes, when average degradation < threshold, after n random intervals
Randomness	Swap SPL threads, swap any threads, swap with thread or empty core

Table 2: Management policy considerations.

for multiple multithreaded fabrics. The objective of these policies is to achieve approximately the same performance as private (per-core) fabrics with the dramatically lower area and power consumption afforded by multithreaded fabrics. When multiple sequential and parallel applications that are compiled to use the SPL are simultaneously executing, the SPL Cluster Manager (Figure 1(d)) optimizes overall performance through two inter-dependent mechanisms: (1) *thread assignment* among the clusters, and (2) *spatial partitioning* and *recombination* of the SPL within each cluster.

There are a number of different factors that contribute to the SPL usage characteristics of an application, including the frequency of SPL accesses and the number of rows needed by each optimized function. The applications that are the biggest concern are those that either make frequent accesses to the SPL or require a large number of rows and therefore incur frequent virtualization. Applications that make frequent SPL accesses can be substantially impacted by poor scheduling, whereas applications with significant virtualization can substantially degrade the performance of other applications sharing the same cluster. We implemented and evaluated a wide range of management policies given the considerations listed in Table 2. We limit our discussion to four representative dynamic management algorithms.

Threads can also be assigned to a particular cluster on a CMP statically through the OS scheduler (in fact, we assume some initial static assignment for our dynamic policies). As we show later, the performance of static thread scheduling varies greatly, with slowdowns ranging from less than 1% to 1028% compared to a 12-row private SPL. Moreover, static scheduling requires dependable *a priori* knowledge about the threads and their potential interactions. Finally, many programs go through different phases during execution and their SPL usage can differ substantially in each phase. Static scheduling cannot exploit this dynamic phase behavior.

3.1 Per Interval Thread Assignment Policies

We first investigated a number of policies that determine an assignment of threads to SPL clusters every interval based solely on the performance of the previous interval and make no use of the spatial partitioning capability of the SPL. We found that, although all applications are impacted by poor scheduling choices, certain applications are impacted more than others. In particular, the largest performance losses occur when threads that require large amounts of virtualization share an SPL cluster with those that rely heavily on the SPL. Based on this insight, the best interval-based thread assignment policy that we devised is *Average Row Assignment*. This policy uses the average number of rows used by the functions of a particular thread as an indicator of its degree of virtualization. Functions that require a large number of rows on average will experience more virtualization, assuming the amount exceeds the number of physical rows available. Thread assignment based on the number of rows alone, however, is insufficient; the SPL access frequency should also be taken into account as an indication of how much each thread relies on the SPL. Threads that heavily utilize the SPL are more likely to be degraded by increased wait time to access the fabric.

Average Row Assignment allocates threads to clusters based on the ratio of the average number of rows used by the thread to SPL accesses. Threads with high access rates and low row usage will have small values while threads with infrequent accesses and high row usage will have high values. To assign threads to clusters we use a *split assignment* policy which aims to schedule threads with high and low values on different clusters. The threads are sorted based on the given metric, the first n/c threads are assigned to the first

cluster, the next n/c threads to the second, and so on, where n is the number of threads and c is the number of clusters.

In order to compute the overall metric, each core maintains counters for instructions issued in the last interval. The core also tracks the number of rows required by each SPL instruction. This latter information is stored in the configuration information for each SPL function and is therefore available to the SPL Cluster Manager.

3.2 Composite Thread Assignment / Spatial Partitioning Policies

Average Row Assignment only considers thread assignment. As described previously, each SPL can also be spatially partitioned. This can be useful if SPL instructions are queued for a long time due to virtualization or due to high SPL usage from the number of threads sharing the SPL. Spatial partitioning can reduce stalls due to either of these cases as it reduces the number of threads that share the same SPL partition.

When considering both thread assignment and spatial partitioning, the number of clusters is effectively dynamic, as each SPL can be divided in half, in quarters, or in one half and two quarters, and thread assignment must account for this cluster size variability. As before, per-core metrics are gathered to determine how to assign threads to however many clusters currently exist. Moreover, the SPL Cluster Manager must determine when to split and merge SPL partitions in each cluster.

To determine thread-to-cluster assignments, this policy, henceforth referred to as Composite, uses the same SPL access to average row ratio with split assignment used by the Average Row scheduler. To determine when to split an SPL cluster, each core tracks the number of cycles an SPL instruction is stalled in the SPL queue and the number of its SPL instructions that are issued. If any thread spends too long on average waiting to issue an SPL instruction, i.e., the average wait time exceeds a threshold, then the SPL is split. Similarly, to determine when to merge, each cluster tracks the number of threads whose average wait time is less than a second threshold. If the sum of this value for the two clusters is greater than the current number of cores sharing a single cluster, then the two clusters are merged. Neither a split nor a merge will occur if both split and merge requests are received for the same cluster in a given interval.

3.3 Learning-Inspired SPL Cluster Management

The policies discussed thus far create their mappings of threads to clusters based solely on the relative ranking of some statistics for each thread during the last interval. Although this generally leads to good mappings, it may not produce the best possible mapping. In an attempt to achieve the best – or at least a better – mapping, we apply machine learning techniques to our cluster mapping problem.

3.3.1 Hill Climbing

Since we desire a fast, purely online approach, we focus on hill climbing. Previous work by Choi and Yeung [10] investigated hill climbing for SMT resource allocation among concurrently running threads. Our scheduling problem is significantly different, and arguably harder, as we have to deal with both resource partitioning and determining which threads should share those partitions.

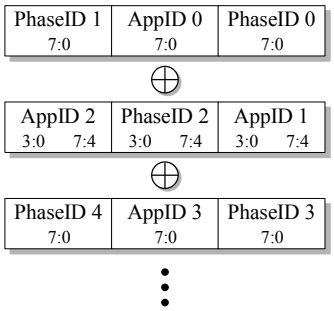


Figure 3: Hash function for phase IDs.

At each scheduling interval the manager may perform one of the following actions: (a) swap two threads from different SPL clusters; (b) split or merge an SPL cluster that is not already at its minimum or maximum size; (c) create a random number of partitions as well as a random mapping of threads to those partitions. The last option adds an element of stochasticity which aims to escape local minima. Each of these options is selected with a predefined probability. We investigate a variety of different restrictions on thread swapping, from allowing only threads using the SPL to be swapped, to allowing any threads to be swapped, to allowing a single thread to be swapped into a cluster with an unused core.

The new assignment is run for the next interval. At the end, the performance of the interval is compared to the performance of the best interval to date for the current phase. If the new mapping achieves better performance, then it is set as the new best mapping; otherwise, the mapping reverts to the previous best mapping. In either case, a new local search step is applied to the current best mapping. After some number of consecutive unbeneficial steps, the best schedule is assumed to have been found and the phase is declared stable. After this point all future intervals in this phase use this stable mapping.

To identify phases, we developed a multi-threaded/multi-programmed workload phase tracker. We use the phase tracker of Sherwood et al. [33] to identify phases for each thread. The phase tracker reports the current phase for each running thread based on the mix of instructions executed during the last phase interval. This phase information is combined to index into a global management history table, which contains the best mapping executed so far for the given set of phases. In order to create a reasonably sized index to access the history table, we developed a hash function to map the application and phase IDs of all currently running threads to a reasonable number of bits. This function takes three byte groups of phase and application IDs (where each phase or application ID is one byte) and XORs them together as shown in Figure 3. The IDs are ordered by application ID. The IDs within every other group are rotated by four bytes to increase diversity. This hashing scheme produces less than a 3% average false match rate for our workloads, and more importantly, degrades performance over a perfect hashing scheme by less than 0.1%.

To determine the relative performance of different mappings, the manager tracks the peak number of instructions graduated by each thread on a per phase basis and calculates the performance degradation for the current phase relative to this peak performance. The performance degradation of all threads are averaged to produce the overall degradation

for the interval. The peak instruction count is determined by averaging the five highest observed graduation rates in that phase.

As will be shown in Section 5, our best hill climbing algorithm is able to match the performance of the Composite manager, but rarely exceeds it. This is due to the large performance degradation that can occur during some of the exploration intervals, and so any slight improvement in mapping over that found by the Composite algorithm is offset by the degradation incurred during the exploration period. Unlike typical pipeline resource allocation, small changes in the thread to SPL assignment can substantially change performance. This effect not only makes finding the optimal mapping difficult, it also significantly degrades throughput (by up to an order of magnitude) during intervals with poor mappings. If too many of these poor mappings are explored, performance degrades severely. Due to these large jumps, the exploration space is not necessarily nicely hill shaped; it is not only quite "bumpy" but there are likely to be numerous local minima that may be difficult to escape.

3.3.2 Hybrid Heuristic-Hill Climbing Manager

Based on our experience with the previous techniques, and additional experimentation, we devised a hybrid manager that addresses three main sources of performance degradation of the prior approaches. The first two sources are present in the heuristic techniques and the last appears with Hill Climbing. First, most programs experience different phases in their execution, during which their use of the shared SPL may vary significantly. As such, the best mapping for one phase may be suboptimal for another, and reaching a new stable mapping may take multiple intervals using the aforementioned interval-based policies. Second, even within a phase there can be a small amount of variability in the performance of a thread. This variation can lead to a ping-ponging effect where threads are constantly being swapped between two clusters. This is especially true for multithreaded workloads where multiple threads can be performing the same task and performance can vary slightly depending on interactions with memory and other threads. This constant swapping can degrade performance due to the overhead for context switching threads. Finally, as mentioned previously, excessive exploration of the assignment space can degrade performance due to the significant performance degradation experienced in certain assignments. To address these issues, we devise a new algorithm that we call *Hybrid Heuristic-Hill Climbing (H3C)* that combines elements of the previous two approaches and incorporates a stability threshold such that further changes are not made if the performance is within some margin of "optimal."

H3C evaluates performance and maintains current assignments using the same phase-based approach as Hill Climbing. Unlike Hill Climbing, no change is made to the assignment for the next interval if the previous interval is determined to be stable. An interval is considered stable if the average performance degradation (as indicated by the graduation rate relative to peak, same as Hill Climbing) for all threads is less than some threshold.

When not stable, the assignment for the next interval is determined by one of two methods. During the first x intervals of a particular multithreaded/multiprogrammed phase the threads are assigned using the Composite algorithm from Section 3.2. The goal of this step is to create a generally

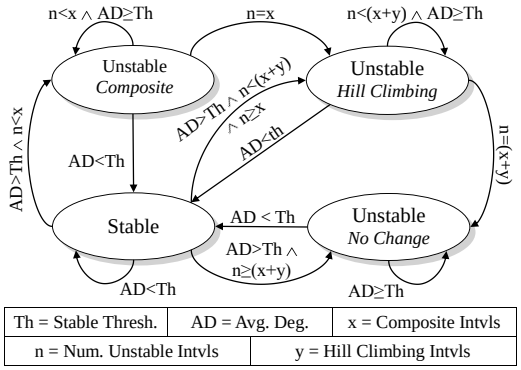


Figure 4: H3C Cluster Manager.

good mapping that can be fine tuned in the next step. During the next y intervals a learning-based local search like that described in Section 3.3.1 is used to try to improve upon the mapping produced by the Composite algorithm. After this step it is assumed that the "best" mapping has been found and no further exploration is performed for this phase, even if the average degradation is not less than the stable threshold in some future intervals. The complete set of H3C state transitions are shown in Figure 4. At each interval, the management table keeps track of the best mapping found so far and H3C reverts back to that mapping as a starting point for the next management interval if the previous interval did not improve upon the performance.

4. EVALUATION METHODOLOGY

We use a highly modified version of SESC [32] to evaluate our proposed multithreaded SPL cluster management policies. We assume processors implemented in 65 nm technology running at 2.0 GHz with a 1.1V supply voltage. The major architectural parameters are shown in Table 3. We use Wattch, Cacti, and HotLeakage to model power.

To model the overhead associated with performing thread management, instruction fetch for all cores is stopped for 1000 cycles at the end of each interval. This value was determined by executing code approximating the scheduling algorithms on our simulator to get an accurate cycle estimate. After this period, the instructions for any threads being migrated are drained and execution is stopped for an additional 500 cycles (again determined by running the requisite code in the simulator) to model the time necessary to context switch all state – including internal SPL state – to the new core. Finally, the threads are started on their new cores, where warm-up of caches and TLBs is modeled. The processor undergoes a similar sequence when the SPL is spatially split or merged by the manager, although in this case the context switch and cache and TLB warm-up are not needed as threads continue to execute on the same core. We experimentally determined the best parameters for the dynamic policies, which are shown in Table 4.

4.1 Phase Tracking

We use the same parameters for our phase tracker as Sherwood et al. [33] with the exception of the phase interval length. We use a smaller 1 million instruction interval due to the shorter phases of some of our applications. Given these parameters, we estimate that the tracker would require around 1 kB of storage per core. Actual phase changes in the program as detected by the phase tracker may not exactly

Branch Predictor	gshare + bimodal
BTB Size	512B
RAS Entries	32
Fetch/Rename Width	2
Issue/Retire Width	1
Integer Registers	64
FP Registers	64
Retire Width	1
Integer Queue Entries	32
FP Queue Entries	16
ROB Entries	64
Int ALUs	1
Branch Units	1
Int Mult/Div Units	1
FP ALU Units	1
LD/ST Units	1
L1 Inst Cache	8kB 2-way
L1 Data Cache	8kB 2-way
L1 Access Latency	2 cycles
L2 Cache	1MB per core
L2 Access Latency	10 cycles
Coherence Protocol	MESI
Main Memory Access Time	100 ns
Phase History Entries	256

Table 3: Architecture parameters.

Policy	Parameter	Value
Composite	Split threshold	16×avg. rows used by core
Composite	Merge threshold	2×avg. rows used by core
Hill Climbing	Probability of splitting SPL	20%
Hill Climbing	Probability of merging SPL	20%
Hill Climbing	Probability of random mapping	10%
Hill Climbing	Threads to consider swapping	All
H3C	Intervals of Composite Scheduling	5
H3C	Intervals of Hill Climbing	5
H3C	Stability threshold	Avg. Deg. $< 4\%$
All	Interval granularity	100k cycles

Table 4: Parameters for dynamic management policies.

	SPL Functions	Max Rows	% Optimized Exec Time	% Dyn. SPL Insts	SPL Usage
300.twolf	1	21	32.7%	0.10%	3.8%
456.hmmer	1	10	85.0%	1.15%	40.2%
462.libquantum	1	11	40.1%	2.19%	13.5%
473.astar	1	2	33.7%	0.79%	2.7%
cjpeg	5	21	49.9%	1.22%	20.6%
MPGenc	4	16	69.1%	0.72%	17.2%
MPGdec-o0	5	20	44.8%	0.35%	15.3%
MPGdec-o3	12	20	47.8%	0.57%	19.3%
crypt	1	298	97.9%	4.48%	99.9%

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

coincide with management interval boundaries as management intervals are based on *cycles* whereas phase tracking is based on *instructions*.

The global SPL Cluster Management history table contains 256 entries. Each entry contains the phase IDs for each thread, the mapping of threads to clusters, and the size of each cluster. We estimate this table would require 4 kB worth of storage.

4.2 Benchmarks

We create four workload mixes to evaluate the performance and power efficiency of our approach. Each workload consists of a combination of parallel and sequential benchmarks. These mixes reflect the type of workloads systems are apt to see in the future as different applications are likely to be parallelized to different degrees. We choose three single threaded benchmarks from SPEC2006 [37], one from SPEC2000 [36], and one from MediaBench [29]. Our multithreaded workloads consist of two benchmarks from ALPBench [30] and a version of the JavaGrande [34] *crypt* benchmark ported to C++. We run the ALPBench version of *MPGdec* with two different command line parameters (-o0 and -o3) as they produce different execution characteristics. Specifically, the o3 version enables additional processing which makes use of the SPL, leading to increased overall SPL usage. A complete list of the benchmarks as well as their SPL usage characteristics can be found in Table 5. Table 6 lists the benchmarks in each workload mix.

In order to create SPL mappings, we profile each benchmark to determine which functions consume the largest portion of total execution time. Following this, we examine each of the functions in order to determine which ones can be efficiently mapped to the SPL. Configurations are then created for those functions by hand, although previous work has shown that compilers can produce good mappings for reconfigurable architectures [2, 4, 45].

Name	Benchmarks
Mix A	MPGenc-2, MPGdec-o0-2, crypt-2, 456.hmmer, 473.astar
Mix B	MPGdec-o3-2, crypt-2, 456.hmmer, 300.twolf, 473.astar, 462.libquantum
Mix C	MPGdec-o3-4, crypt-2, 300.twolf, 462.libquantum
Mix D	MPGenc-4, crypt-2, cjpeg, 462.libquantum

Table 6: Workload composition. For parallel workloads the number after the name indicates the number of threads spawned during the run.

Since dynamic thread scheduling is most useful when applications experience phase changes, we need to run the benchmarks long enough to witness these phase changes. The best option is to run benchmarks to completion. Due to the long running time of SPEC benchmarks with reference inputs, however, we are only able to run our non-SPEC benchmarks to completion. For our SPEC benchmarks we use Early SimPoints [31] to select two 250 million instruction SimPoints from the original source code (i.e., code not utilizing the SPL). Since using the SPL changes the number of instructions executed, we determine where each of the two SimPoints begin and end and augment the code to fast forward through all but these two intervals. In this way both the original and SPL versions of the code execute functionally equivalent amounts of code. We select relatively long

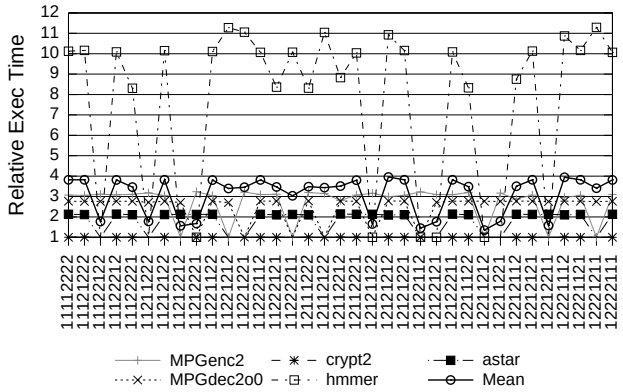


Figure 5: Performance of Mix A for multithreaded SPL clusters with all possible static schedules relative to private 12-row SPL.

intervals to capture phase changes within an interval. In order to make our comparison fair, we continuously respawn threads that finish early so that longer running threads still experience contention for the SPL due to the shorter running threads. We stop the simulation when the longest running benchmark completes and report the execution time for each benchmark averaged over all completed runs.

5. RESULTS

We first motivate the need for dynamic thread assignment and spatial partitioning by showing the varied performance achieved with static thread assignment relative to large private SPLs. We then compare dynamic management to the best, worst, and median-case static assignments. We also compare our approach with the performance and energy×delay^2 that would be ideally achieved by replacing the SPL with additional cores.

5.1 Static Assignment Performance

The OS scheduler could statically assign threads to clusters (i.e., maintain the schedule throughout execution) using information regarding expected SPL usage gleaned from the compiler. For each workload, we simulate all 35 possible static assignments, and extract the best, worst, and median static assignments based on the mean relative execution time of all benchmarks. This information tells us what an oracle static scheduler could achieve, the worst performance that could occur if SPL usage is not taken into account by the scheduler at all, and the margin for error, i.e., whether most schedules are closer to the best or the worst schedule.

Figure 5 shows the performance of each benchmark for one of the workloads for all static assignments (results for the other three workloads show similar overall trends). The labels on the x-axis indicates the cluster, 1 or 2, to which each thread is assigned. A label of 12112212, for example, indicates the first spawned thread is assigned to cluster 1, the second thread to cluster 2, the third thread to cluster 1, etc. The performance is highly variable, varying by as much as 1028% between the best and worst static schedules for some benchmarks. The mean performance for the best, worst, and median static schedules for each workload relative to the 12-row private SPL baseline is shown in Figure 6 (second, third, and fourth bars). Individual benchmark performance for each workload is shown in Figure 7[3]. While in some cases the best (oracle) static scheduler performs reasonably well, the results for the worst schedule indicate that a static scheduler that is oblivious to SPL usage may perform poorly relative to the private 12-row SPL organization. Moreover, Figure 6 shows that the median schedule is much closer to the worst case schedule than the best case schedule. Thus, there is little margin for error in static scheduling; such errors could easily arise due to the lack of static information regarding the fabric contention among applications.

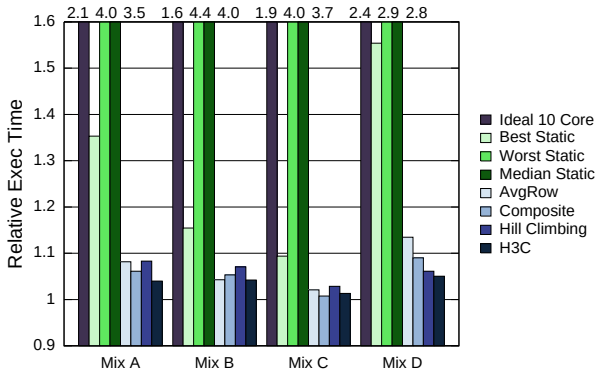


Figure 6: Average execution time for each workload relative to 12-row private SPL.

5.2 Performance of Dynamic SPL Cluster Management

The individual benchmark and average workload performance of the four representative dynamic management policies presented in Section 3 relative to the performance of private 12-row SPL is shown in Figures 7 and 6, respectively.

5.2.1 *Average Row and Composite Policies*

Overall, the Average Row and Composite policies outperform the best possible static assignment by 21.9% and 23.6%, respectively. The benefits of permitting the manager to control spatial partitioning as done in the Composite policy are demonstrated by comparing the overall results for both policies (Figure 6). Compared with the much higher overhead private 12-row SPL organization, the Average Row approach experiences a 7.0% slowdown and the Composite policy experiences a 5.3% slowdown.

For a few of the benchmarks, the dynamic algorithms occasionally improve performance relative to the 12-row private SPL. This occurs because we simulate private L2 caches. When scheduled on several different cores, threads may make use of multiple L2 caches. Thus, on an L2 cache miss, the data might be sourced from another L2 cache rather than the slower main memory. To ensure that this effect is not the primary reason for the improvement of our policies, we ran tests where all L2 misses are forced to access main memory. We found that the cache "sharing" effect on performance was negligible in comparison to the effect of the Cluster Manager.

5.2.2 *H3C Policy*

As mentioned previously, and shown in the results, the Hill Climbing manager typically does not outperform the simpler Composite approach due to the performance loss in-

[3]Note that the best schedule is based on mean performance, and as such might not be best for an individual benchmark.

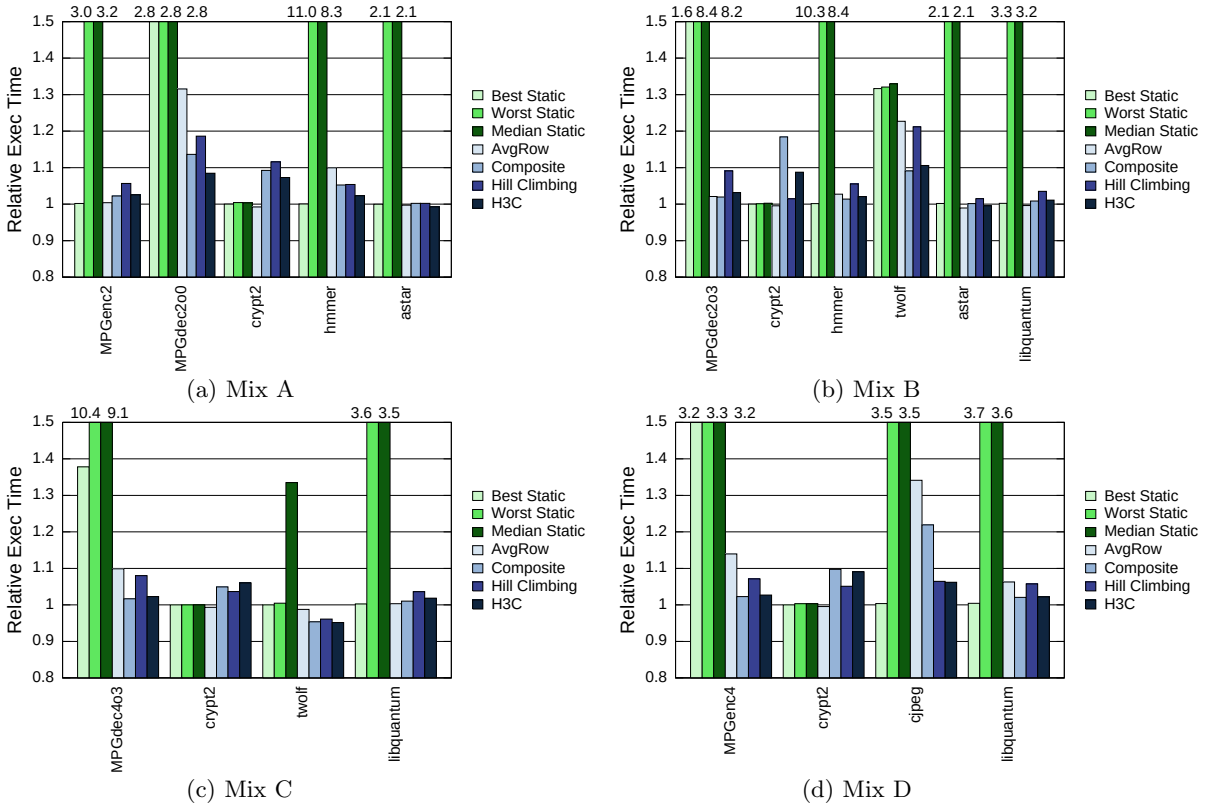


Figure 7: Performance of multithreaded SPL clusters with dynamic scheduling algorithms and best, worst, and median static schedules, relative to private 12-row SPL.

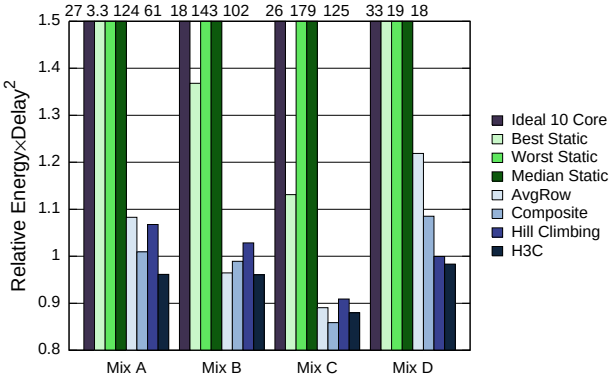


Figure 8: Average energy×delay^2 for each workload relative to 12-row private SPL.

curred during exploration. The H3C manager achieves the best all around performance, outperforming all other options in all but one case. In the one exception, Mix C under the Composite manager, the performance with H3C is less than 1% worse than the Composite scheduler. Overall the H3C policy achieves 25.3% better performance than the best static schedule. Compared to the 12-row private SPL, the H3C management approach experiences only a 3.6% slowdown while consuming 4X less area.

When energy×delay^2 (ED^2) is considered (Figure 8), the benefits of multithreaded SPLs incorporating both scheduling and spatial partitioning are further accentuated. The H3C manager achieves 5.4% better ED^2 than the 12-row private baseline on average (again with a 4X lower area cost due to the shared fabrics). By contrast, the best static schedule experiences an average 179% worse ED^2 than the 12-row private SPL. H3C is the only approach that provides better ED^2 than the 12-row baseline for all four workloads.

Another benefit of the dynamic policies is fairness. For most of the best static schedules, some of the threads achieve near optimal performance while others experience significant slowdown. With the dynamic policies, the performance impact is quite uniform across the threads.

H3C Component Analysis

In Section 3.3.2 we detailed a number of factors that limit the performance of the Composite and Hill Climbing managers and how the H3C manager incorporates techniques to address these issues. To assess the importance of each, and also confirm that the proposed solutions achieve their stated goals, we run simulations where one or more of H3C features are modified. In particular we look at cases where the stability threshold is eliminated (No Stability), where hill climbing is eliminated (No Hill), and where additional hill climbing is performed (Extra Hill). We also look at a case where Composite scheduling is performed at every interval (essentially adding phase information to the base Composite manager) (Composite+Phase).

Figure 9 shows the performance loss of each case relative to the H3C manager. The H3C manager outperforms all of these alternatives in every instance. This confirms that hill climbing, stability detection, and phase analysis are all important and that eliminating any one of them degrades the quality of the manager. The most important factor is the stability threshold, without which performance degrades by

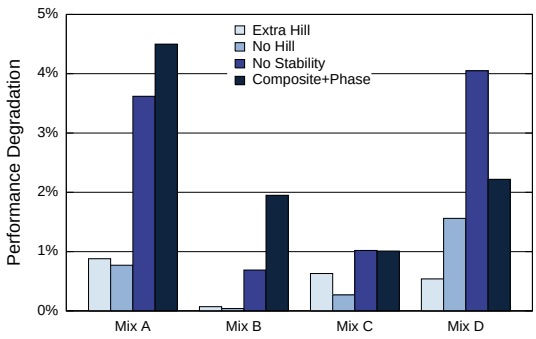


Figure 9: Performance degradation relative to H3C.

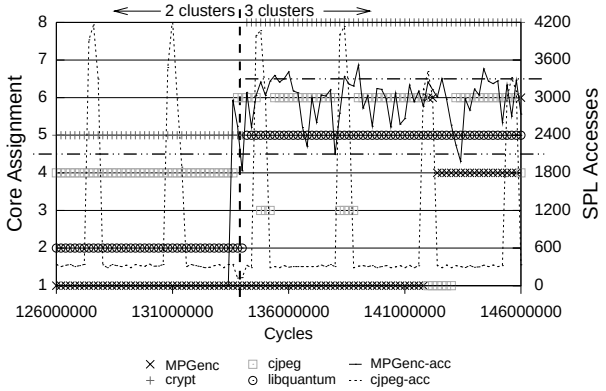


Figure 10: Thread-to-core assignment and SPL accesses for Mix D with H3C.

2.3%. Continuously running the Composite scheduler with phase information leads to a similar losses. Both indicate the benefits of limiting unnecessary exploration.

Phase Analysis

One of the key features of our management schemes is their ability to dynamically adapt to different application phases. Figure 10 shows an example of the thread scheduling and cluster partitioning for a section of Mix D with H3C management. The graph shows the thread-to-core assignment for the four main threads along with the SPL access patterns of the two threads that change phases during the given period. The horizontal dotted lines in the graph show which cores share a SPL partition and the vertical line indicates when one of the clusters is partitioned.

At the start of the example, four threads are actively using the SPL. The two *crypt* threads share one cluster and the two single threaded applications share the other in order to minimize conflicts. Around 134M cycles, *MPGenc* starts a section that uses the SPL. The H3C manager monitors SPL usage and determines how to schedule threads and partition the fabric to adapt to this change. In particular, one of the clusters is divided so that *crypt* still has its own partition, and the assignment of threads to clusters is rearranged based on current usage statistics.

The figure also shows how the manager can adapt to the changing access pattern of *cjpeg*. During phases when its access rate increases, *cjpeg* is rescheduled on the larger cluster to achieve better performance. Unlike the Composite manager, however, which always makes the same change, we can see in the last two access peaks for *cjpeg* that the H3C manager explores other options in an attempt to find an even better mapping.

5.2.3 Replacing the SPL with Additional Cores

Figures 6 and 8 also show results for each workload in which each SPL is replaced by one additional single issue core; in other words, the workloads are run on the conventional cluster on the right side of the chip diagram of Figure 1(a). These results were obtained by simulating a given workload using the original benchmarks (no SPL code) with eight cores (one per thread), and then ideally scaling the results to 10 cores. This ideal scaling is achieved by linearly reducing the execution time by 1/5, but optimistically increasing the energy by only 12.5% (even though there are now 25% more cores).

A comparison of the Ideal 10 Core and all of the dynamic scheduling results in these graphs substantiates previous work that demonstrated significant benefits with SPL on particular applications. Performance degrades by 62-143% when the workloads are run on the 10-core cluster rather than the two with multithreaded SPL, and the ED^2 differences are even more pronounced: up to 34X worse ED^2 for Ideal 10 Core. We emphasize again that on a large-scale CMP, those applications that are not compiled to use the SPL can be scheduled on a non-SPL cluster. For those threads that are compiled to use the SPL, effective assignment of threads to multithreaded SPL clusters, coupled with judicious dynamic spatial partitioning of the SPLs, is crucial to achieving good performance, power efficiency, and area efficiency.

6. RELATED WORK

6.1 Reconfigurable Processors

Several excellent survey papers [23, 41] provide an overview of the contributions of prior reconfigurable computing projects. The bulk of these efforts focus on a single processor core with an attached reconfigurable fabric. There is a dearth of prior work in addressing how reconfigurable logic can best benefit future CMPs. In this section, we focus on those ideas most related to our proposed thread assignment and spatial partitioning policies.

Caspi et al. [6] propose SCORE, a reconfigurable system which uses a stream programming model. Their design incorporates a single CPU and multiple reconfigurable blocks. Configurations can map to these blocks both spatially and temporally. A number of research efforts [5, 13, 20] have investigated the high level integration of a reconfigurable fabric on-chip. All of these, however, only investigate the integration with a single core, although Garcia and Compton [20] state that their technique could be extended to a multicore system.

In [21], configuration data for a reconfigurable coprocessor is shared among multiple cores in order to increase fabric utilization by allowing a larger number of configurations to coexist in the fabric. Chen et al. [8] investigate the benefits of including reconfigurable ISA support in a multicore processor and find that combining program parallelization with custom ISA support provides larger speedups than the product of the two techniques applied in isolation.

Our previous work [43] identifies a number of characteristics of past reconfigurable proposals that are found to be highly amenable to incorporating reconfigurable fabrics in CMPs. We designed a shared SPL based on these fea-

tures, and analyzed the impact of incorporating the fabric with processors of different complexity. While the emphasis of [43] is on the fabric design, this paper proposes a complete hardware/software approach to managing multithreaded SPL clusters in future CMPs, including spatial partitioning and thread scheduling policies.

Reconfigurable computing has recently been gaining increased attention from industry. Both Intel and AMD permit tighter integration of FGPAs with general purpose processors through HyperTransport, QuickPath, and licensing of front side bus technology [18, 19]. Convey Computer's HC-1 pairs an Intel processor with a reconfigurable coprocessor and allows different instruction sets to be loaded into the coprocessor [12].

6.2 Thread Scheduling and Dynamic Resource Sharing

The benefits of dynamic thread scheduling in small scale CMP/SMT systems has been explored for a number of purposes, including cache-aware scheduling [17, 26, 40], thermal management [11], and SMT resource-aware scheduling [16, 35]. Most of these efforts deal with temporally scheduling threads between time slices where the number of threads is greater than the number of processor contexts. They aim to minimize contention or maximize sharing between threads scheduled in the same interval. Our work is different in a number of aspects, including that we perform spatial partitioning with all threads running at the same time.

Numerous SMT resource management techniques exist that aim to either directly [7, 10] or indirectly [15, 42] control the amount of processor resources that any thread consumes. These techniques aim to maximize the benefit each thread realizes from its share of resources, particularly by limiting threads with outstanding misses from hogging resources. With SPL resource management, on the other hand, threads that are "hogging" the SPL are actually the ones receiving the most benefit from the fabric, and so SPL management is not as simple as just limiting threads that use a lot of the SPL. Also, unlike front-end resources, which can be reallocated quickly at a fine granularity, there is nontrivial overhead involved in both supporting and dynamically switching between different SPL sharing degrees, which impacts the techniques that can be employed.

Previous research has proposed sharing other architectural components among multiple cores. Several efforts investigate how to best allocate shared L2 cache space among multiple threads [9, 25, 38]. Sun's UltraSPARC T1 [39] shares a single floating point unit between its eight SMT cores. Kumar et al. [27] investigate sharing FP units, crossbar ports, and L1 instruction and data caches between two cores. Their work focuses on temporal sharing, and does not consider dynamic spatial techniques such as splitting a cache in half if inter-thread conflicts are too high. We also propose more advance policies that combine machine learning, phased-based analysis, and stability control. Prior work in optimizing resource allocation during different phases [14, 33] only address single applications. In our multithreaded environment, each thread has its own current phase and we must deal with optimizing thread assignment and resource allocation as phases change across multiple applications.

Our situation is more difficult than any of this previous work as we must concurrently manage both cluster thread assignment and fabric partitioning.

7. CONCLUSIONS

We propose dynamically managed multithreaded reconfigurable fabrics for future CMPs. We examine a range of dynamic management policies that vary in their approach mapping threads to clusters, as well as how they exploit the ability to spatially partition each SPL to mitigate inter-thread conflicts.

Of the four representative approaches that we present, our best policy judiciously combines elements of machine learning, phase-based analysis, and stability detection to assign threads to SPL clusters and spatially partition the SPLs on-the-fly. This approach outperforms an oracle static scheduler, and experiences only a small slowdown compared with much larger private SPLs dedicated to each core. We also demonstrate dramatic improvements over allocating the SPL area to additional cores. Overall, we demonstrate that sharing reconfigurable fabrics and managing their resources on-the-fly are key to making reconfigurable fabrics an attractive, cost-effective, option for future CMPs.

ACKNOWLEDGMENTS

This research was supported by an NSF Graduate Research Fellowship; NSF grants CCF-0916821, CCF-0811729, and CNS-0708788; and equipment grants from Intel.

8. REFERENCES

[1] Advanced Micro Devices. AMD Athlon X2 Dual-Core Details. `http://www.amdcompare.com/us-en/desktop/details.aspx?opn=ADH2350IAA5DD`, 2007.

[2] M. Budiu and S. C. Goldstein. Fast Compilation for Pipelined Reconfigurable Fabrics. In *Proc. 1999 ACM/SIGDA 7th Int'l Symposium on Field Programmable Gate Arrays*, pages 195–205, Feb. 1999.

[3] S. Cadambi, J. Weener, S. Goldstein, H. Schmit, and D. Thomas. Managing Pipeline–Reconfigurable FPGAs. In *Proc. 1998 ACM/SIGDA 6th Int'l Symposium on Field Programmable Gate Arrays*, pages 55–64, Feb. 1998.

[4] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33:62–69, Apr. 2000.

[5] J. Carrillo and P. Chow. The Effect of Reconfigurable Units in Superscalar Processors. In *Proc. 2001 ACM/SIGDA 9th Int'l Symposium on Field Programmable Gate Arrays*, pages 141–150, 2001.

[6] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE). In *Proceedings of the 10th Int'l Workshop on Field-Programmable Logic and Applications*, pages 605–614, Aug. 2000.

[7] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically Controlled Resource Allocation in SMT Processors. In *Proc. IEEE/ACM 37th Annual Int'l Symposium on Microarchitecture*, pages 171–182, 2004.

[8] Z. Chen, R. N. Pittman, and A. Forin. Combining Multicore and Reconfigurable Instruction Set Extensions. In *Proc. 18th ACM/SIGDA Int'l Symposium on Field Programmable Gate Arrays*, pages 33–36, 2010.

[9] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proc. IEEE/ACM 39th Annual Int'l Symposium on Microarchitecture*, pages 455–468, Dec. 2006.

[10] S. Choi and D. Yeung. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In *Proc. 33rd*

IEEE/ACM Int'l Symposium on Computer Architecture, pages 239–251, 2006.

[11] T. Constantinou, Y. Sazeides, P. Michaud, B. Fetis, and A. Seznec. Performance Implications of Single Thread Migration on a Chip Multi-Core. *SIGARCH Computer Architecture News*, 33(4):80–91, 2005.

[12] Convey Computer. The Convey HC-1 Computer. White Paper, Nov. 2008.

[13] 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*, pages 980–985, 2003.

[14] A. S. Dhodapkar and J. E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *Proc. 29th IEEE/ACM Int'l Symposium on Computer Architecture*, volume 30, pages 233–244, 2002.

[15] A. El-Moursy and D. H. Albonesi. Front-End Policies for Improved Issue Efficiency in SMT Processors. In *Proc. 9th IEEE Symposium on High Performance Computer Architecture*, page 31, 2003.

[16] A. El-Moursy, R. Garg, D. Albonesi, and S. Dwarkadas. Compatible Phase Co-Scheduling on a CMP of Multi-threaded Processors. In *Proc. of the 20th Int'l Parallel and Distributed Processing Symposium*, page 10, 2006.

[17] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX 2005 Annual Technical Conference*, pages 395–398, Berkeley, CA, 2005.

[18] M. Feldman. FPGA Acceleration Gets a Boost from HP, Intel. *HPCWire*, Sept. 2007.

[19] M. Feldman. Reconfigurable Computing Prospects on the Rise. *HPCWire*, Dec. 2008.

[20] P. Garcia and K. Compton. A Reconfigurable Hardware Interface for a Modern Computing System. In *Proc. 2007 IEEE Symposium on Field–Programmable Custom Computing Machines*, pages 73–84, April 2007.

[21] P. Garcia and K. Compton. Kernel Sharing on Reconfigurable Multiprocessor Systems. In *Int'l Conference on Field Programmable Technology*, pages 225–232, 2008.

[22] S. Goldstein, H. Schmit, M. Moe, M. Budiu, and S. Cadambi. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *Proc. 26th IEEE/ACM Int'l Symposium on Computer Architecture*, pages 28–39, May 1999.

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

[24] *Intel Core 2 Extreme Processor X6800 and Intel Core 2 Duo Desktop Processor E6000 and E4000 Sequences*, 2007. Intel Datasheet: 313278-004.

[25] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proc. 13th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[26] P. Koka and M. H. Lipasti. Opportunities for Cache Friendly Process Scheduling. In *Workshop on Interaction Between Operating Systems and Computer Architecture*, 2005.

[27] R. Kumar, N. Jouppi, and D. Tullsen. Conjoined-core Chip Multiprocessing. In *Proc. IEEE/ACM 37th Annual Int'l Symposium on Microarchitecture*, pages 195–206, 2004.

[28] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer–Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[29] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A Tool for Evaluation and Synthesizing Multimedia and Communications Systems. In *Proc. IEEE/ACM 30th Int'l Symposium on Microarchitecture*, pages 330–335, 1997.

[30] 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 Int'l Symposium on Workload Characterization*, pages 34–45, 2005.

[31] E. Perelman, G. Hamerly, and B. Calder. Picking Statistically Valid and Early Simulation Points. In *Proc. 12th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, Sept 2003.

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

[33] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proc. 30th IEEE/ACM Int'l Symposium on Computer Architecture*, pages 336–349, June 2003.

[34] 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)*, pages 8–17, 2001.

[35] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proc. 9th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[36] Standard Performance Evaluation Corporation. SPEC CPU Benchmark Suite. `http://www.specbench.org/cpu2000/`, 2000.

[37] Standard Performance Evaluation Corporation. SPEC CPU Benchmark Suite. `http://www.specbench.org/cpu2006/`, 2006.

[38] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *J. Supercomputing*, 28(1):7–26, 2004.

[39] Sun Microsystems, Inc. *UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005*, Mar. 2006.

[40] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2007 Conference on EuroSys*, pages 47–58, 2007.

[41] 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*, 152(2):193–205, March 2005.

[42] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. 23rd IEEE/ACM Int'l Symposium on Computer Architecture*, pages 191–202, 1996.

[43] M. Watkins, M. Cianchetti, and D. Albonesi. Shared Reconfigurable Architectures for CMPs. In *Proc. of the 18th Int'l Conference on Field-Programmable Logic and Applications*, Sept. 2008.

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

[45] Z. A. Ye, N. Shenoy, and P. Banerjee. A C Compiler for a Processor with a Reconfigurable Functional Unit. In *Proc. 2000 ACM/SIGDA 8th Int'l Symposium on Field Programmable Gate Arrays*, pages 95–100, Feb. 2000.