# Erasing Core Boundaries for Robust and Configurable Performance[*]

Shantanu Gupta Shuguang Feng Amin Ansari Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{shangupt, ansary, shoe, mahlke} @umich.edu

## ABSTRACT

Single-thread performance, reliability and power efficiency are critical design challenges of future multicore systems. Although point solutions have been proposed to address these issues, a more fundamental change to the fabric of multicore systems is necessary to seamlessly combat these challenges. Towards this end, this paper proposes CoreGenesis, a dynamically adaptive multiprocessor fabric that blurs out individual core boundaries, and encourages resource sharing across cores for performance, fault tolerance and customized processing. Further, as a manifestation of this vision, the paper provides details of a unified performance-reliability solution that can assemble variable-width processors from a network of (potentially broken) pipeline stage-level resources. This design relies on interconnection flexibility, microarchitectural innovations, and compiler directed instruction steering, to merge pipeline resources for high single-thread performance. The same flexibility enables it to route around broken components, achieving sub-core level defect isolation. Together, the resulting fabric consists of a pool of pipeline stage-level resources that can be fluidly allocated for accelerating single-thread performance, throughput computing, or tolerating failures.

## Keywords

Multicores, Performance, Reliability, Reconfigurable Architectures

## 1. INTRODUCTION

As a result of growing power, thermal and complexity concerns in monolithic processors, major hardware vendors have lead a migration to multicore processors composed of relatively simple cores. Today, the sizes of multicores vary from 2-6 cores in desktop systems to 16-64 cores in throughput-oriented computing domains, e.g. SUN Niagara [1], Intel Larrabee [2] and Tilera TILE64 [3]. Despite having been attenuated, several challenges pertaining to performance, power and reliability still remain in the multicore paradigm. First, multiple cores are effective for throughput computing, but they provide little to no gains for sequential applications. Even if a major transition towards parallel programming occurs in the future, Amdahl's law dictates that the sequential component of an application will present itself as a performance bottleneck. Second, power constraints limit the number of cores / resources that can be kept active on a chip, motivating the need for customized and power-proportional processing [4]. Finally, the increasing vulnerability of transistors with each passing generation [5, 6], jeopardize the objective of throughput sustainability over the lifetime of a multicore chip.

In this landscape of multicore challenges, prior research efforts have focused on addressing these issues in isolation. For example, to tackle single-thread performance, a recent article by Hill and Marty [7] introduces the concept of *dynamic multicores* (Figure 1(a)) that can allow multiple cores on a chip to work in unison while executing sequential codes. This notion of *configurable performance* allows chips to efficiently address scenarios requiring throughput computing, high sequential performance, and anything in between. Core Fusion [8], Composable Lightweight Processors [9] and Federation [10] are representative works with this objective. However, the scope of present day *dynamic multicore* solutions is limited as they cannot provide customized processing, as in [4, 11], or better throughput sustainability, as achieved by techniques in [12, 13]. The customized processing in [4] (Figure 1(b)) is typically accommodated by introducing heterogeneity of types and number of functional units, execution models (in-order, OoO), etc., into different cores. Whereas, better throughput sustainability can be provided by fine-grained reliability solutions like CCA [12] and StageNet [13], that disable broken pipeline stages, instead of entire cores (Figure 1(c)), within a multicore.

Unfortunately, by virtue of being independent efforts, combining existing performance, power and reliability solutions for multicores is neither cost-effective nor straightforward. The overheads quickly become prohibitive as the changes required for each solution are introduced, with very little that can be amortized across multiple techniques. Configurable performance requires dedicated centralized structures (adding drawbacks such as access contention/latency, global wiring), customization requires a variety of static core designs, and fine-grained reliability requires either large amounts area for cold spares or the flexibility to share resources across cores. Apart from excessive overheads, a direct attempt to combine these solution also faces engineering hurdles. For instance, when combining CoreFusion [8] (a configurable performance solution) and StageNet [13] (a fine-grained reliability solution), two prominent issues arise: 1) CoreFusion requires centralized structures for coordinating fetch, steering, commit across fused pipelines. These structures become single points of failure and limit reliability benefits of StageNet. 2) StageNet requires a decoupled microarchitecture for its sub-core defect tolerance. This is not compatible with CoreFusion, as resources within a single CoreFusion core are tightly coupled together.

Instead of targeting one challenge at a time, the goal of this paper is to devise a design philosophy that can naturally be extended to handle a multitude of multicore challenges seamlessly, while overlapping costs, maintaining efficiency and avoiding centralized structures. Towards this end, this paper proposes the CoreGenesis (CG) architecture (see Figure 1(d)), an adaptive computing substrate that is inherently flexible, and can best align itself to the immediate system needs. CG eliminates the traditional core boundaries and organizes the chip multiprocessor as a dynamically con-

[*]To appear in the 43rd International Symposium on Microarchitecture, December 2010.

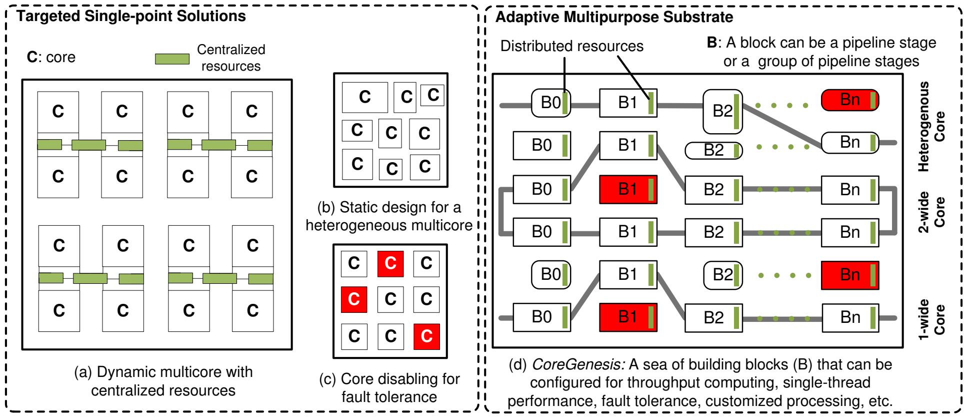

Targeted Single-point Solutions
C: core
Centralized resources
(a) Dynamic multicore with centralized resources
(b) Static design for a heterogeneous multicore
(c) Core disabling for fault tolerance
Adaptive Multipurpose Substrate
Distributed resources
B: A block can be a pipeline stage or a group of pipeline stages
B0
B1
B2
Bn
Heterogenous Core
2-wide Core
1-wide Core
(d) CoreGenesis: A sea of building blocks (B) that can be configured for throughput computing, single-thread performance, fault tolerance, customized processing, etc.

Figure 1: Contemporary solutions for multicore challenges (a,b,c) and vision of this work (d). In (a), centralized resources are used to assist in fusing neighboring cores. In (b) and (d), different shapes/sizes denote heterogeneity. In (c) and (d), dark shading marks broken components.

figurable network of building blocks. This sea of building blocks can be symmetric or heterogeneous in nature, while varying in granularity from individual pipeline stages to groups of stages. Further, the CG pipeline microarchitecture is decoupled at block boundaries, providing full flexibility to construct logical processors from any complete set of building blocks. Another key feature of the CG proposal is the use of distributed resources to coordinate instruction execution across decoupled blocks, without any significant changes to the ISA or the execution model. This is a major advancement over prior configurable performance works, and addresses the shortcomings of centralized resources.

Resources from CG's sea of blocks can be fluidly allocated for a number of performance, power and reliability requirements. Throughput computing can be optimized by forming many single-issue pipelines, whereas sequential performance can be accelerated by forming wider-issue pipelines. Power and performance characteristics can be further improved by introducing heterogeneous building blocks in the fabric, and appropriately configuring them (dynamically or statically) for active program phases or entire workloads. This enables a dynamic approach to customized processing. Finally, fault tolerance in CG can be administered at the block granularity, by disabling the broken components over time.

Guided by this architectural vision, in this paper, we present a CG instance that targets configurable performance and fine-grained reliability. For the fabric, an in-order pipeline model is used with single pipeline stages as its building blocks. As a first step, we define mechanisms for decoupling pipeline stages from one another (inspired by the StageNet architecture [13]). This enables salvaging of working stages from different rows of the fabric to form logical processors, thereby tackling the throughput sustainability challenge. To address configurable performance, we generalize the notion of logical processors to form processors of varying issue widths.

The engineering of distributed resources to support the assembly of decoupled pipeline stages into a wide-issue processor is especially hard due to the heavy co-ordination and communication requirements of an in-order superscalar. Our solution adopts a best effort strategy here, speculating on control and data dependencies across pipeline ways, and falling back to a light-weight replay in case of a violation. To register these violations, hardware schemes were formulated for distributed control, register and memory data flow management. The frequency of data flow violations from instructions executing on two different pipeline ways was found to be a leading cause of performance loss. We address this by incorporating compiler hints for instruction steering in the program binary. This circumvents the hurdles in fusing in-order cores, as presented in [14], while also achieving a near-optimal pipeline way assignment. Overall, the manifestation of CG presented in this paper relies on interconnection flexibility, microarchitectural innovations, and compiler directed instruction steering, to provide a unified performance-reliability solution.

## 2. RELATED WORK

Within the framework of multicore chips, efficient solutions that can deliver configurable performance and throughput sustainability are desirable. This section gives an overview of prior works targeting these issues. Table 1 summarizes the key aspects of CG in comparison to the relevant prior proposals. CG stands out by simultaneously offering configurable performance and fine-grained reliability while eliminating centralized structures. This section also presents a study that motivates a need for unified performance-reliability solutions for the sake of efficiency.

### 2.1 Single-Thread Performance Techniques

**Dynamic multicores.** Dynamic multicore processors consists of a collection of homogeneous cores that can work independently to provide throughput computing, or a subset of them can be fused together to provide better single-thread performance. Core Fusion [8] is a dynamic multicore design that enables the fusion of adjacent OoO cores to form wider-issue OoO processors. Federation [10], on the other hand, combines neighboring in-order cores to form an OoO superscalar. Both these approaches employ centralized structures (for fetch management, register renaming, instruction steering, etc.) to assist in aggregation of pipeline resources. In contrast, Composable Lightweight Processors (CLP) [9] leverages the EDGE ISA and compiler support to eliminate centralized structures, enabling it to scale up to 64-cores. CG also eliminates centralized structures, but its compiler support is limited to generating hints for instruction steering, and ISA is modified to include this hint carrying instruction. Multiscalar [15] is a seminal work that can compose a large logical processor from many smaller processing elements. It uses an instruction sequencer to distribute task sub-graphs among the processing elements, and relies on hardware to satisfy dependencies. However, in all these prior schemes, resources within individual cores are tightly coupled together, dismissing the opportunity for fine-grained reliability.

Another distinction of CG is that it fuses in-order pipelines to form wider-issue in-order processors. While out-of-order fusion provides opportunities for hiding latency (large instruction window

Table 1: Comparison to Prior Work

| | Configurable Performance | Fine-grained Reliability | No centralized structures | Supports in-order model | Supports heterogeneity |
|---|---|---|---|---|---|
| CG (this paper) | ✓ | ✓ | ✓ | ✓ | ✓ |
| CLP [9] | ✓ | | ✓ | ✓ | |
| Core Fusion [8], Federation [10], Multiscalar [15] | ✓ | | | | |
| StageNet [13], CCA [12] | | ✓ | ✓ | ✓ | |
| Heterogeneous CMPs [4] | | | ✓ | ✓ | ✓ |

sizes), in-order fusion is made harder due to the negligible room for inefficiency. In fact, Salverda et al. [14] argue that in-order pipeline fusion is impractical because of the associated hardware overheads for interleaving active data flow chains (instruction steering). CG circumvents these challenges by using compiler hints to guide instruction steering, and employing simple mechanisms to detect and recover from data flow violations.

**Heterogeneous CMPs.** Heterogeneous designs exhibit good power and performance characteristics for their targeted class of applications. However, being a static design, its effectiveness is limited outside this set or when flexibility is desired. For instance, in a scenario where all applications prefer throughput computing, a heterogeneous CMP will operate sub-optimally.

In addition to static scheduling of jobs on heterogeneous CMP cores, there have also been dynamic scheduling approaches to match program phase behaviors to cores. Core contesting [11] is one such example, but it runs the same program redundantly on different cores to allow a faster transfer of state between them. In CG, inclusion of heterogeneous blocks can allow static, dynamic as well as fine-grained dynamic exploitation of program phase to architecture mapping. This is possible due to CG's inherent flexibility to swap resources between pipelines.

**Clustered Architectures.** The early research in clustered architectures was to enable wider issue capabilities, without adding sophisticated hardware support. The Multicluster [16] architecture is a good example of this, and it uses static instruction scheduling from compile time. CG, on the other hand, uses a compiler clustering algorithm [17] to generate hints that are used for dynamic instruction steering. This is also in contrast to past works that solely use hardware support [18] to implement heuristics for distributing instructions among clusters in a superscalar.

## 2.2 Multicore Reliability Solutions

**Coarse-Grained Reconfiguration.** High-end server systems, like Tandem NonStop and IBM zSeries [19], typically rely on coarse grained spatial redundancy to provide a high degree of reliability. However, dual and triple modular redundant systems incur significant overheads in terms of area and power, and cannot tolerate a high failure rate. More recently, ElastIC [20] and Configurable Isolation [21] proposed disabling of broken cores in a chip multiprocessor. Architectural Core Salvaging [22] is an interesting approach that continues to use broken cores for a subset of computation, while the results are not impacted by the failures. Although good in a limited failure rate scenario, all of these proposals need a massive number of redundant cores, without which they face the prospect of rapidly declining processing throughput as faults lead to core disabling.

**Fine-Grained Reconfiguration.** A newer category of techniques use stage-level reconfiguration (isolates broken stages, not cores) for reliability. StageNet (SN) [13] groups together a small set of pipelines stages with a simple crossbar interconnect. By enabling reconfiguration at the granularity of a pipeline stage, SN can tolerate a considerable number of failures. Romanescu et al. [12] also propose a similar multicore architecture, Core Cannibalization Architecture (CCA), that exploits stage level reconfigurability. However, CCA allows only a subset of pipelines to lend their stages to other broken pipelines, thereby avoiding full interconnection. In CG, fine-grained reconfiguration is supported in the same way as SN.

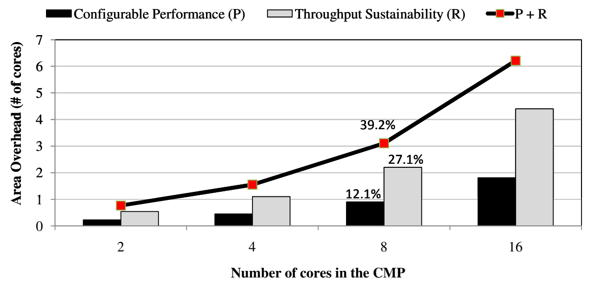


Figure 2: Area overhead projections (measured as number of cores) for supporting configurable performance (P) and throughput sustainability (R) in different sized CMP systems. P+R curve shows the cumulative overhead. For this plot, throughput sustainability is defined as the ability to maintain 50% of original chip's throughput after three years of usage in the field.

## 2.3 Combining Performance and Reliability

All prior works target the multicore challenges separately, either configurable performance or throughput sustainability (reliability). The central problem here is that solutions for each of these require new hardware to be incorporated into existing CMPs. This turns out to be an expensive proposition, as the hardware costs are additive. We conducted a small study to assess this cost. Figure 2 shows the results from this study using Core Fusion [9] as the configurable performance solution, and standard core disabling for throughput sustainability. The line plot shows the cumulative overhead of performance (Core Fusion) and reliability (core disabling) solutions (P+R). Resulting overhead is almost 40% additional area. There are two factors at play here: 1) costs are additive, as the two solutions share nothing in common, 2) reliability is administered at core level (instead of being fine-grained). On top of this, the design, test, verification and validation efforts need to be duplicated for performance and reliability separately. The next section presents CG, our unified performance-reliability solution, that overcomes these issues to a large extent.

## 3. THE COREGENESIS ARCHITECTURE

### 3.1 Overview

The manifestation of CoreGenesis (CG) architecture presented here is a unified performance-reliability solution that allows fusion of standalone cores for accelerating single-thread performance as well as isolation of defective pipeline stages for sustainable throughput. The CG fabric consists of a large group of pipeline stages connected using non-blocking crossbar switches, yielding a highly configurable multiprocessor fabric. These switches replace all direct wire links that exist between the pipeline stages including the bypass network, branch mis-prediction signals and stall signals. The pipeline microarchitecture within CG is completely *decoupled*, and all pipeline stages are standalone entities. The symmetric crossbar interconnection allows any set of unique stages to assemble as a logical pipeline. As a basis for the CG design, an in-order core is used[1], consisting of five stages namely, fetch (F), decode (D), issue (I), execute/memory (E/M) and writeback [23]. Figure 3 shows the arrangement of pipeline stages across four interconnected cores and a conceptual floorplan of an 8-core CG chip. Note that all modifications introduced within CG are limited to the core microarchitecture, leaving the memory hierarchy (private L1 / unified L2) untouched. Further, the caches are assumed to have their own protection mechanism (like [24]), while CG tolerates faults within the core microarchitecture.

In Figure 3, despite having one stage failure (shaded) per core, CG is able to salvage three working pipelines. Further, given a set of active threads, CG can judiciously allocate these pipeline resource to them in proportion to their instruction level parallelism. For instance, in the figure, thread 1 (low ILP) is allocated one pipeline and thread 2 (high ILP) is allocated the remaining two pipelines.

Configurable performance helps CG in dealing with the software diversity present in modern day CMP systems. It keeps pipelines separate for throughput computing and dynamically configures two (or more) pipelines into a multi-issue processor for sequential workloads. This morphing of individual pipelines into a *conjoint processor* requires no centralized structures, maintains reliability benefits, and is transparent to the programmer. In Figure 3, CG processor 2 is an example of a conjoint processor assimilated using two pipeline stages of each type. As part of a conjoint processor, the two pipelines cooperatively execute a single thread. The instruction stream is fetched alternately by the two pipelines - odd numbered ops by one pipeline and the even numbered ops by the other. All instructions are tagged with an *age* to maintain the program order during execution and instruction commit. Regardless of where an instruction is fetched, it can be executed on either of the two pipelines depending upon its source operands. We refer to this as *instruction steering*. An instruction executing on the same pipeline that fetches it is said to be straight-steered, while that executing on some other pipeline is said to be cross-steered. This dynamic instruction steering is performed with an objective of minimizing data dependency violations, and is critical for achieving true multi-issue performance. CG employs a compiler level analysis for statically identifying data dependency chains (Section 3.5) and the issue stage applies this knowledge (during run-time) to steer instructions to the most suitable pipeline.

The natural support for fine-grained reconfiguration allows CG to achieve its second objective of throughput sustainability. For instance, in Figure 3, CG is able to efficiently salvage the working stages from the pool of defective components to form functional processors. By the virtue of losing resources at a smaller granularity, isolation of broken pipeline stages reaps far better rewards than traditional core disabling. To realize its reliability benefits, the CG system relies on a fault detection mechanism to identify broken stages and a software configuration manager to consolidate the working ones (by reprogramming the crossbars). Fault detection can be achieved using a combination of manufacture-time and in-field periodic testing, details of which are beyond the scope of this paper.

[1]Deeper and more complex pipelines can be segmented at logical boundaries of elementary pipeline stages (F,D,I,E,W) to benefit from the CG approach.

Table 2: CoreGenesis (CG) challenges. The challenges can be classified on the basis of single and conjoint pipeline configurations. The check marks (✓) are used for solutions that were straightforward extension of prior work on decoupled architectures. Whereas the question marks (?) are open problems that are solved in this paper.

| | Control flow | Register data flow | Memory data flow | Instruction steering |
|---|---|---|---|---|
| Single pipeline | ✓ | ✓ | N/A | N/A |
| Conjoint pipelines | ? | ? | ? | ? |

## 3.2 Challenges

Although the performance and reliability benefits of its configuration flexibility are substantial, there are a number of hurdles faced by the CG architecture. There are four principal challenges, and they span correctness and performance issues for both single pipeline processors as well as conjoint pipelines processors:

**Control flow management:** The decoupled nature of the CG pipeline makes global signals such as pipeline flush and stall infeasible. In the context of a single pipelines, the control flow management is crippled by the absence of a global flush signal. The problem is even more severe in the case of conjoint processors. Pipeline fetch stages need to read complementary instructions from a single program stream, and make consistent decisions about the control flow (i.e., whether to take a branch or not).

**Register data flow management:** Back-to-back register data dependencies are typically handled by the operand bypass network, which relies on timely inter-stage communication. Unfortunately, the decoupled design of CG pipelines makes the bypass network impractical. In the case of conjoint processors, this problem is further aggravated by the presence of cross pipeline register dependencies. The decentralized instruction execution needs a mechanism to track dependencies, detect violations, and replay instructions for guaranteeing correctness.

**Memory data flow management:** Memory instructions are naturally serialized in the case of a single pipeline CG processor, as all of them reach the same memory stage. However, similar to register data flow violations, memory data flow violations can also occur between pipelines of a conjoint processor, leading to a corruption in global state.

**Instruction steering:** In a conjoint processor, issue stages have the option to straight steer the instructions to same pipeline or cross steer it to the other pipeline. This decision has to be dynamically made for every instruction such that the number of cross pipeline data dependencies is minimized. A recent study by Salverda et. al [14] establishes that steering is central to the challenge of in-order pipeline fusion, and further concludes that a hardware-only steering solution is impractical.

Table 2 summarizes all the challenges in the context of single and multiple pipelines working as a logical processor. A subset of these challenges have been solved (marked with a ✓) by a prior work, StageNet(SN) [13]. SN is a decoupled pipeline microarchitecture for fine-grained fault tolerance. The interconnection bandwidth solution from SN is generic and applies to both single/conjoint scenarios.

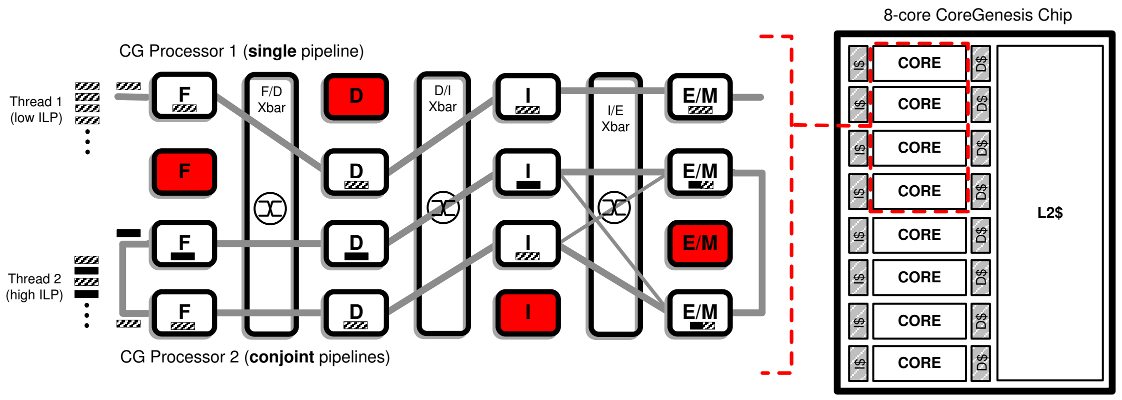


Figure 3: An 8-core CoreGenesis (CG) chip with a detailed look at four tightly coupled cores. Stages with permanent faults are shaded in red. The cores within this architecture are connected by a high speed interconnection network, allowing any set of stages to come together and form a logical processor. In addition to the feed-forward connections shown here, there exist two feedback paths: E/M to I for register writeback and E/M to F for control updates. In CG processor 2 (conjoint pipelines), instructions (prior to reaching E/M stage) can switch pipelines midway, as a result of dynamic steering.

The control, register data flow, memory data flow, and instruction steering solutions for conjoint processors are contributions of this paper (marked with a *?*). All of these are new mechanisms, and were made harder by the fact that unlike a true multi-issue machine (and even Core Fusion [8]), CG does not have centralized structures, and needs to get performance by combining very loosely coupled resources. For the sake of completeness, in the descriptions that follow, we also provide a quick overview of the solutions for single pipeline case from [13].

## 3.3 Microarchitectural Details

This section describes microarchitectural changes needed by the CG architecture, a majority of which are clever tricks to detect control and data flow violations in a distributed fashion. The relatively complex task of instruction steering is off-loaded to the compiler (Section 3.5).

### 3.3.1 Control Flow

**Single Pipeline.** For a single pipeline CG processor, the absence of a global pipeline flush signal complicates the control flow management. In the event of a branch mis-prediction, the decoupled pipeline needs a mechanism to squash the instructions fetched along the incorrect path. The introduction of a 1-bit stream identification (SID) to all the in-flight instructions targets this problem [13]. The basic idea is to use the SID for distinguishing instructions on the correct path from those on the incorrect path. The fetch and the execute stages maintain single bit SID registers, both of which are initialized to the same value (the discussion here is simplified, the actual scheme adds a SID register to every stage). The fetch SID is used to tag all incoming instructions. And, the execute stage matches an instruction's SID tag against the execute SID before letting it run. If at any point in time, a branch instruction is resolved as a mis-prediction by the execute stage, the execute SID is toggled and the update is sent to the fetch stage. All in-flight instructions that are tagged with the stale SID are recognized (by the execute) to be on the incorrect path and are systematically squashed over time. In parallel to this squashing, after receiving the branch update from the execute, the fetch toggles its own SID and starts fetching correct path instructions. Note that a single bit suffices here because the pipeline execution model is in-order and can have only one resolved branch mis-predict outstanding at any given time (since all instructions following it become invalid).

Table 3: Control cases. Each case represents a pair of consecutive program instructions in a 2-issue conjoint processor. The first and second rows in this table show the instructions fetched in the leader and follower pipelines, respectively.

| **Case 1** branch not taken | **Case 2** branch not taken | **Case 3** branch taken | **Case 4** branch taken |
|---|---|---|---|
| OP | BR | OP | BR |
| BR | OP | BR | OP |

**Conjoint Pipelines.** In a dual-pipeline conjoint processor, one pipeline is designated as the *leader* and other as the *follower*. To balance the usage, both pipelines fetch alternate instructions from the program stream, i.e., if leader fetches from $PC$, follower fetches from $PC + 4$. The logical program order is maintained by tagging every instruction with a unique (monotonically increasing) `age` tag. Fetch stages are augmented with age counters (offset by 1) that are incremented in steps of two whenever an instruction is fetched and tagged. Thus, the leader pipeline will tag instructions with ages 0, 2, 4, and so on; and follower will tag them with ages 1, 3, 5, and so on. By virtue of interleaving program counter values, both pipelines together fetch the complete program stream and record the program order in the `age` tags. These tags are later used by the execute $\rightarrow$ issue crossbar (EI xbar) to commit instructions in the program order.

The above description of distributed fetch works fine until a branch instruction is encountered. For proper operation, CG needs a decentralized control handling mechanism that keeps both pipelines in sync when making a control decision. The control flow can encounter four distinct cases shown in Table 3.

Cases 1 and 2 are the most straightforward ones, as the branch is not taken. Both pipelines continue as normal as the branch has no impact on the control flow. For case 3, we need both pipelines to take the branch simultaneously. This can be achieved if their branch predictors completely mirror each other and same address lookup is performed by both pipelines. We maintain this mirroring by sending all branch prediction updates (from execute/memory stage) to both fetch stages. For consistent look-ups, the leader pipeline addresses its branch predictor using $Leader\_PC{+}4$, and the follower addresses it using $Follower\_PC$ (and by design $Follower\_PC = Leader\_PC + 4$). As both the predictors are synchronized, they will return the same prediction and target address. Finally, for case 4, we again need both pipelines to take the branch. In addition to the mechanism for case 3, the follower pipeline must also invali-

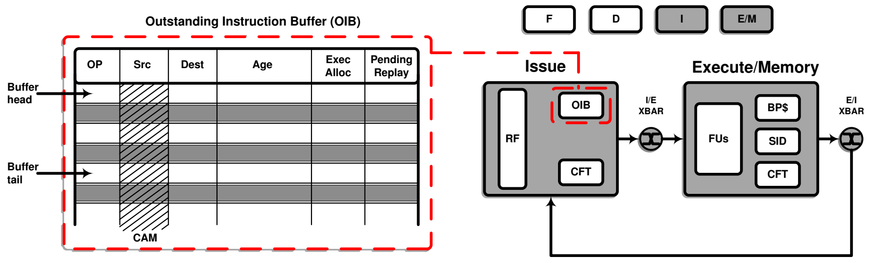


Figure 4: CG pipeline back-end with structures for detecting register data flow violations and initiating replays. The outstanding instruction buffer (OIB) and current flow tag (CFT) registers are the two additions for conjoint processors. Also shown here is the bypass cache (BP$) for data forwarding within a single pipeline.

date its $OP$ which is on the wrong path. A simple logic is added to the decode stage to carry this out. The decode stage invalidates any operation that is 1) in the follower pipeline *and* 2) is predicted as a taken branch by the fetch *and* 3) is not a real branch instruction.

In the case of a branch mis-predict, the squashing of instructions for conjoint processors is a direct extension of the SID scheme presented for single pipelines. In conjoint processors, both pipelines maintain a single logical value for the SID, and all branch resolution updates are sent back concurrently to the fetch stages.

### *3.3.2 Register Data Flow*

**Single Pipeline.** The data forwarding within a single pipeline can be emulated using a small *bypass* cache in the execute stage. The key idea is to use this bypass cache for storing results from recently executed instructions, and supplying them to later instructions. The experiments in [13] show that a bypass cache that holds last six results is sufficient.

**Conjoint Pipelines.** For conjoint processors, the data flow management gets involved due to the distributed nature of execution. The instructions are issued and executed on different pipelines, and cross-pipeline register data dependencies can occur frequently (instruction fetched by pipeline X, but needs register produced by pipeline Y). In an ideal scenario, we would like issue stages to always steer the dependent instructions to the execute which most recently produced the source values. More of this discussion on instruction steering follows later in Section 3.5. Nevertheless, in a practical design, the steering mechanism is bound to make some mistakes as each pipeline's issue stage has incomplete information about the in-flight instructions. Our solution, in a nutshell, is to have each pipeline maintain a local version of the outstanding data dependencies, and monitor write-backs by the other pipelines to detect any data flow violations that might have occurred. Upon detecting such a violation, a replay is initiated.

The first requirement for data flow management is proper maintenance of the register file. The register files for all pipelines (that constitute a conjoint processor) are kept coherent with each other. This is achieved by sending all register write-backs to both issue stages simultaneously, similar to the way Alpha 21364 [25] kept its two clusters consistent. Further, the write-backs from the two pipelines are serialized by the network interface between the execute and the issues stages. The crossbar switch prioritizes the write-back based on the age tag of the instructions, maintaining correct program commit order. This way, cross-pipeline data dependencies, which are sufficiently far away in the program, go through the register file. However, all the instructions that are issued before their producers have written back to the register file remain vulnerable to undetected data flow violations.

To catch such undetected data flow violations, each pipeline can track locally issued (in-flight) instructions and monitor the write-backs to detect any data dependency violations. We accomplish this using a new structure in the issue stage named *outstanding instruction buffer* (OIB) (see Figure 4). The OIB is similar in concept to the reorder buffer in an OoO processor. However, it is much smaller in size, and needs to store only 5 (pipeline depth from issue to write-back) instructions in the worst case. Each instruction entry in the OIB stores: (1) op code, (2) sources, (3) destination, (4) age tag, (5) execute stage allocation (execute stage where the instruction was steered), and (6) pending replay bits (one per source operand). The OIB behaves as a CAM for its second field (instruction source). Pending replay bit for a source operand denotes whether it can cause a data flow violation. Instructions are inserted into the OIB at the time they are issued. At the time of an instruction write-back, following actions take place:

- The destination value ($R_{dest}$) of the instruction writing-back ($I_{wb}$) updates the register file. The corresponding OIB entry for $I_{wb}$ is also freed.
- $R_{dest}$ is used to do a CAM look-up in the OIB. This returns any in-flight instruction ($I_{in\_flight}$) that uses $R_{dest}$ as a source.
- If $I_{in\_flight}$ was sent to the same execute stage where $I_{wb}$ executed, then the bypass cache would have successfully forwarded the register value. The pending replay bit is reset (to 0) for this source operand of $I_{in\_flight}$.
- If $I_{in\_flight}$ was sent to some other execute stage, then a data flow violation is possible and the pending replay bit for this source is set (to 1).

Over time, the replay bit for a source operand can get set/reset multiple times, with the final write to it made by the closest producer operation for every consumer. If an issue stage receives a write-back for an instruction with a pending replay bit set for any of its source operands, it implies that the producer of value(s) for this instruction has executed on an execute stage different from where this instruction was steered. And, therefore, a data flow violation has occurred. A replay is initiated at this point (replay mechanism is discussed later in this section).

### *3.3.3 Memory Data Flow*

To provide correct memory ordering behavior in a conjoint pipelines processor, we use a local store queue in the issue stages that monitors load operations performing write-back for store-to-load forwarding violations, and a speculative store buffer in the execute/memory stage to allow delayed release of memory store operations (to save against accidental memory corruption). Note that cache hierarchy is left unmodified in CG. L1 caches are private to the pipelines, single ported and naturally kept coherent by standard cache coherence protocols.

Figure 5 shows the back-end of a CG pipeline with an emphasis on structures needed for proper memory handling. A store buffer

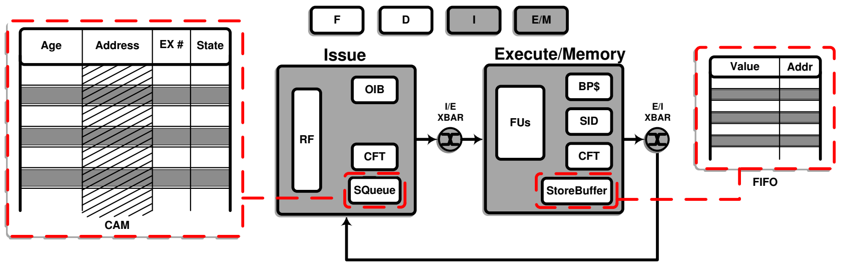


Figure 5: CG pipeline back-end with an emphasis on structures added for handling memory data flow violations.

(*StoreBuffer*) is added to the execute/memory stage to hold onto the store values before they are released to the memory hierarchy. Although a common structure in many processors, in CG, the store buffer also serves the purpose of keeping speculative stores from corrupting memory state. A store queue (*SQueue*) is added to the issue stages to tabulate the outstanding store instructions, and their present states. Every store instruction can have two possible states. All issued store instructions are entered into the local store queue and get into the *store sent* state. Write-back for this store instruction confirms that it is not on an incorrect execution path. At this point, a *pseudo commit* signal is sent (over the same crossbar switch) to the execute/memory stage that executed this store, and the store instruction state becomes *pseudo commit sent*. Upon receiving this signal, the execute releases the store value at the head of the store buffer to the memory. This way, only stores on the correct path of execution update the memory. There are three possible cases involving the memory operations that need a closer scrutiny (see Table 4).

Table 4: Memory flow cases. Each case represents a pair of instructions that are flowing together in a 2-issue conjoint processor.

| | Case 1 | Case 2 | Case 3 |
|---|---|---|---|
| Leader pipeline | $BR$ (mis-predicted) | $ST_1$ | $ST$ |
| Follower pipeline | $ST$ | $ST_2$ | $LD$ |

In case 1, a mis-predicted branch occurs right before a store in the program order. Since this store is already executed by the execute/memory stage, its value is entered into the store buffer. Fortunately, in accordance to the commit order, the branch operation writes back before the store. Thus, the store never gets to write-back and does not release a pseudo commit for itself. Eventually this store is removed from the store buffer when the mis-predicted branch flushes the pipeline. In case 2, the pseudo-commit is released for $ST_1$ before $ST_2$. Thus, to the memory hierarchy, the correct ordering is presented. In case 3, when the load is about to commit, both issue stages check if any of the outstanding stores conflicts with this load (using the store queues). If there is indeed such a store that precedes the load in the program order (based on age), and was sent to a different execution stage, then a replay is initiated starting from this load.

### 3.3.4 Replay Mechanism

The replay mechanism adds a single bit of state in the issue and execute/memory stage called the *current flow tag* (CFT), and leverages the OIB in the issue stage for re-streaming instructions (see Figure 4). The CFT is a single bit (similar to the SID for branches) to identify the old (wrong) instructions from the new (replaying) instructions in the back-end. All issued instructions are tagged with the CFT bit. The *head* and the *tail* pointers in the OIB mark the window of in-flight instructions, which are replayed in the event of a register or memory data flow violation. The violation is first identified by any one issue stage, which consequently sends out a *flush* instruction to both execute stages. This flips the CFT bit, resets the bypass cache and clears the store buffer. Following this, other issue stages are sent replay signals, and all of them start re-issuing instructions from their respective OIBs (starting at the head pointer) and tagged with an updated CFT bit. The old instructions, tagged with a stale CFT, are uniformly discarded by both issues during the write-back.

## 3.4 Interconnection

CG interconnection network is a simple, one-hop connection. It employs bufferless, non-blocking crossbars to connect adjacent levels of pipeline stages. This allows all pairs of stages, that share a crossbar, to communicate simultaneously. As an interface to the interconnection network, pipeline stages maintain a latch on both inputs and outputs. This makes the interconnection network a separate stage, and thus, it does not interfere with critical paths in the main processor stages.

In order to make the basic crossbar design suitable for the CG architecture, three features are required:

**Multicast:** The CG depends on the capability of the interconnection to send one value to multiple receivers. For instance, write-backs are sent to both issue stage register files simultaneously.

**Instruction steering:** CG requires capability to steer instructions from issue to the appropriate execute stage. A single (header) bit in the instruction payload is added to specify the output (execute stage) an instruction wants to reach.

**Age prioritization:** In the case of write-backs, older instructions have to be given priority. This requires an addition to the router to let it prioritize packets (instructions in our case) on the basis of their age.

Crossbar switch fabrics with crosspoints can support multicast by setting the crosspoint gate logic to high for multiple outputs. A recently proposed SRAM based crossbar architecture, named XRAM [26], demonstrates this ability with a low power and area overhead. The instruction steering and age prioritization can be added in the wrapping logic around the crossbars. However, the XRAM paper suggests that these features can also be implemented using circuits.

**Crossbar reliability, power and timing:** In order to protect the interconnection network, fault tolerant version of the crossbars are used in CG. This is similar to the approach in [27]. The interconnection power can be broken into crossbar power and interconnection link power. Both of these are accounted for in our evaluations, as per the methodology in [28]. The absence of buffers in our network significantly cuts down on this overhead. And finally, we model interconnection link latency using intermediate pitch wire model from ITRS 2008 in 65nm technology, and make sure that it does not exceed critical paths of pipeline stages.

## 3.5 Instruction Steering

CG depends upon intelligent steering of instructions between conjoint pipelines in order to minimize performance degradation from data dependency replays. The instruction steering decisions need to be made at the time of instruction issue. Broadly speaking, the objectives of instruction steering are two-fold: 1) balance the workload on the two pipelines, and 2) minimize the number of replays. Our experiments showed that using a purely hardware based solution for dynamic steering is neither cheap nor effective for in-order pipeline fusion. This concurs with the conclusion of [14]. Thus, CG adopts a hybrid software/hardware approach for instruction steering. In a nutshell, a compiler pass is used to assign instruction streams to the pipelines. These hints are then encoded into *steering* instructions that are made part of the compiled application binary. The hardware recognizes these special steering instructions and uses them to effectively conduct dynamic steering.

Steering instructions between different pipelines in a conjoint processor is analogous to data-flow graph (DFG) partitioning for clustered VLIWs. The goal is to obtain a balanced workload that takes advantage of hardware parallelism (multiple clusters) and reduces the need for inter-cluster moves (transferring values between clusters). Leveraging generic clustering algorithms to form instruction streams for CG is fairly straightforward. When cross-pipeline dependencies cannot be avoided, the CG equivalent of an inter-cluster move is the replay mechanism described in the previous section. Further, CG's broadcast-based write-back ensures that any dependent instructions that are separated by more than $n$ intervening instructions will not incur a replay even if they are steered to different execute stages, where $n$ is the issue-to-writeback latency. Therefore, the two main objectives of clustering algorithms, minimizing inter-cluster moves and overlapping moves with other computation, naturally result in instruction streams that are amenable to the CG architecture. For our evaluations, we used the well known Bottom-Up Greedy (BUG) [17] clustering algorithm to generate hints for steering.

A *STEER_OP* instruction is introduced in order to encode this compiler-generated steering information. Two such instructions are inserted (for leader and follower pipelines) at the beginning of every instruction block (basic block / super block). *STEER_OP* instructions are simply bit encoding of the pipeline assignment for every instruction within that block (multiple instructions are inserted for large code blocks).

Figure 6 shows an example of the complete hybrid steering setup in action. The first step consists of performing the BUG clustering algorithm in the compiler. The second step encodes the clustering algorithm suggested pipeline assignments and embeds them as *STEER_OP* (top two instructions in the final code, 'L' here stands for leader pipeline assignment and 'F' for follower pipeline assignment). When the leader pipeline fetches its *STEER_OP LFLLF*, it learns the steering directions for instruction 1 (L), 3 (F), 5 (L), 7 (L) and 9 (F). The follower pipeline behaves analogously.

## 4. EVALUATION

## 4.1 Methodology

A comprehensive set of tools are used for the evaluation of CG. The evaluation setup spans program compilation and microarchitecture level simulation, down to area, power and wearout modeling.

Table 5: Architectural parameters.

| Baseline architecture | |
|---|---|
| Pipeline | 4-stage in-order OR1200 RISC [30] |
| Frequency | 400 MHz |
| Area | $0.71mm^2$ (65nm process) |
| Power (baseline OR1200 core) | $94mW$ |
| Branch predictor | Global, 16-bit history, gshare predictor, BTB size - 2KB |
| L1 I$, D$ | 4-way, 16 KB, 1 cycle hit latency |
| L2 $ | 8-way, 64 KB (per core), 5 cycles |
| Memory | 40 cycle hit latency |
| **CG specific parameters** | |
| Interconnection | full non-blocking crossbars, 64-bit wide, bufferless |
| Outstanding instruction buffer (OIB) | 5 entries |
| Store queue, store buffer sizes | 3, 3 |
| Bypass cache size | 6 |

**Compilation for instruction steering.** The Trimaran compilation system [29] is used to perform the BUG clustering algorithm [17] for instruction steering. Inter-cluster move latency of five cycles is used as an input to the algorithm.

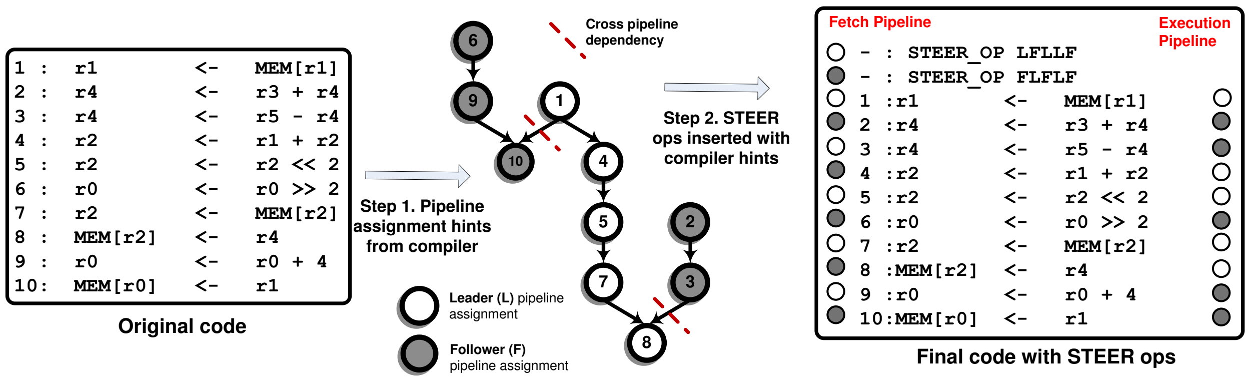
**Microarchitectural simulation.** The microarchitectural simulator for CG models a group of 4-stage in-order pipelines (similar to the OR1200 core [30]) interconnected to form a network of stages. The simulator was developed using the Liberty Simulation Environment [31] from Princeton. The architectural attributes are detailed in Table 5. The L2 cache is unified and its size is $64\,KB \times$ *the number of cores*. The original OR1200 pipeline is also used as the baseline for single-thread performance. The architectural simulations are conducted for benchmarks chosen from three sources: SPEC2000int, SPEC2000fp and multimedia kernels.

**Area overhead (for design blocks and wires).** Industry standard CAD tools with a library characterized for a 65nm process are used for estimating the area of design blocks. A Verilog description for the OR1200 microprocessor was obtained from [30]. Most CG modifications: OIB, SQ, SB, bypass cache, etc., are essentially small memory structures, and their areas are estimated using similar sized CAM structures. All non-memory structures, such as replay logic, stream identification control, and crossbars, are implemented as Verilog modules to obtain accurate area numbers. The area for the interconnection wires between stages and crossbars is estimated using the same methodology as in [32, 8], with the intermediate wiring-pitch (at 65nm) taken from the ITRS road map [33].

**Power and thermal modeling.** Power dissipation for various modules in the design is simulated using Synopsys Primepower an execution trace of OR1200 running media kernels. The crossbar power dissipation was simulated separately using a representative activity trace. The crossbar Verilog was placed and routed using Cadence Encounter before running it through Primepower. The stage to crossbar interconnection power was calculated using standard power equations [28] with capacitance from Predictive Technology Model [34] and intermediate wiring-pitch from 65nm node (ITRS [33]). The thermal modeling was conducted using HotSpot 3.0 [35].

**Wearout modeling.** For wearout modeling, mean-time-to-failure (MTTF) was calculated for various components in the system using the empirical models found in [36]. An entire core was qualified to have an MTTF of 10 years. The calculated MTTFs are used as the mean of the Weibull distributions for generating time to failures (TTF) for each module (stage/crossbar) in the system. For the sake of consistency in comparisons, wearout modeling makes assumptions similar to those in [13].

The MTTF of 10 years was chosen as a rough estimate for the



```
1 :  r1        <-   MEM[r1]
2 :  r4        <-   r3 + r4
3 :  r4        <-   r5 - r4
4 :  r2        <-   r1 + r2
5 :  r2        <-   r2 << 2
6 :  r0        <-   r0 >> 2
7 :  r2        <-   MEM[r2]
8 :  MEM[r2]   <-   r4
9 :  r0        <-   r0 + 4
10:  MEM[r0]   <-   r1
```

```
○ - : STEER_OP LFLLF
● - : STEER_OP FLFLF
○ 1 :r1       <-   MEM[r1]    ○
● 2 :r4       <-   r3 + r4    ●
○ 3 :r4       <-   r5 - r4    ●
● 4 :r2       <-   r1 + r2    ○
○ 5 :r2       <-   r2 << 2    ○
● 6 :r0       <-   r0 >> 2    ●
○ 7 :r2       <-   MEM[r2]    ○
● 8 :MEM[r2]  <-   r4         ○
○ 9 :r0       <-   r0 + 4     ●
● 10:MEM[r0]  <-   r1         ●
```

Original code
Step 1. Pipeline assignment hints from compiler
Cross pipeline dependency
Leader (L) pipeline assignment
Follower (F) pipeline assignment
Step 2. STEER ops inserted with compiler hints
Fetch Pipeline
Execution Pipeline
Final code with STEER ops
6
9
10
1
4
5
7
2
3
8

Figure 6: Instruction steering. The white nodes indicate instructions assigned to the leader pipeline while the shaded nodes correspond to the follower pipeline. The instruction fetch is perfectly balanced between the two pipeline, but the execution is guided by the steering.

future technologies: 22nm and beyond. Note that the 65nm technology node was only used to get power and area overheads for comparisons.

**Quantitative comparison against other schemes.** For experiments involving multicores, CG is compared against two other systems: 1) A conventional CMP chip where, a core is considered to be faulty when any of its modules fail; 2) a SN chip [13], as it shares similarities with CG in the way it tackles reliability.

## 4.2 Single-thread performance

Configurable performance in CG relies on its ability to accelerate single-thread performance by conjoining *in-order* pipelines. Figure 7 shows a plot comparing the performance of four CG configurations normalized to the 1-issue in-order baseline (OR1200). The plot also includes a 2-issue in-order baseline for the sake of comparisons. The CG configurations are expressed as:
number_of_pipelines_conjoint × issue_width_of_pipeline_stages.
The following configurations are examined: 1-issue (1x1) CG single pipeline, 2-issue (2x1) CG conjoint pipelines, 2-issue (1x2) CG single pipeline, and 4-issue (2x2) CG conjoint pipelines.

**Conjoining single-issue stages.** This compares a (1x1) CG pipeline and a (2x1) CG conjoint pipeline against the two baselines. All pipeline stages are inherently single-issue in this set up. The 1-issue CG pipeline (1x1) performs roughly 10% worse than the 1-issue baseline, primarily due to the inter-stage transfer inefficiencies from decoupling (this is similar to results in the SN work [13]). On the other hand, the (2x1) CG conjoint pipeline was found to deliver a consistent performance advantage over the 1-issue baseline, while lagging behind the 2-issue baseline. The gains are most prominent for the SPECfp and kernel benchmarks. In fact, for some of the kernel benchmarks, almost a 2X performance gain was seen while using the conjoint processor. The availability of long and independent data dependence chains in these benchmarks made this result possible.

In contrast, a few of the benchmarks showed negligible to negative performance improvements while using the conjoint processor, namely 176.gcc, 197.parser and 177.mesa. This was due to the lack of independent streams of instructions in these workloads. Instructions in these benchmarks typically formed long dependence chains, and the compiler pass (for steering) ended up allocating most instructions to the same pipeline (to minimize the replay cost). This resulted in a nearly complete serialization of the program, rendering half of the execution resources useless. The few cases where instructions were steered to different pipelines lead to data flow violations and worsened the overall performance by initiating the replay. Barring these three benchmarks, the rest of the results strongly favor the conjoint pipelines CG processor design. On average, a 48% IPC gain is seen over the single pipeline CG processor.

**Conjoining dual-issue stages.** This compares a (1x2) CG single pipeline processor, a (2x2) CG conjoint pipeline processor and a 2-issue baseline processor. All pipeline stages are inherently dual-issue in this set up. A single logical pipeline in this system would behave as a dual-issue processor. Using the CG conjoining principles, any two dual-issue pipelines can be then combined to form a quad-issue CG processor. The (2x2) 4-issue conjoint pipeline shows a 35% improvement in performance over the (1x2) 2-issue pipeline, and a 25% improvement in over the 2-issue baseline. Note that by making the pipeline stages dual-issue, the fault isolation granularity for the system is reduced by half. This discussion is continued later in this section along with the reliability implications.

Our experiments with conjoining more than two pipelines (both single and dual-issue) at a time did not show very favorable results. The two main reasons for this were: 1) the limited availability of independent data-flow chains, and 2) the constraints placed by an in-order issue architecture.

**Replay costs.** The performance advantage of a conjoint CG processor is largely determined by the efficiency of instruction steering in balancing the load between the two pipelines, while minimizing replays. Here, we analyze the cost of these replays in a 2-issue (2x1) CG conjoint pipelines processor. Figure 8 shows three components of the total execution time for all the benchmarks: memory flow (MemFlow) violation replay cycles, register flow (RegFlow) violation replay cycles and normal operation cycles. A majority of the benchmarks devote a small fraction of their execution time to the replay cycles, with an average of 15%. Out of the total replay cycles, memory replay contributes a negligible fraction. This is an expected result because memory replay only happens when a store to load forwarding is missed by the system, which by itself is a rare event for in-order processors. From the perspective of power efficiency, these results are encouraging because only a very small percentage of the work performed by the system goes to waste. Note that a low number of replay cycles does not necessarily imply good benchmark performance. For instance, all instructions in a conjoint processor can be steered to the same pipeline resulting in zero replays (no cross-pipeline dependency). However, no speedup compared to the baseline would be observed.

## 4.3 Energy-efficiency Comparison

Energy-efficiency of designs can be compared using $BIPS^3/watt$

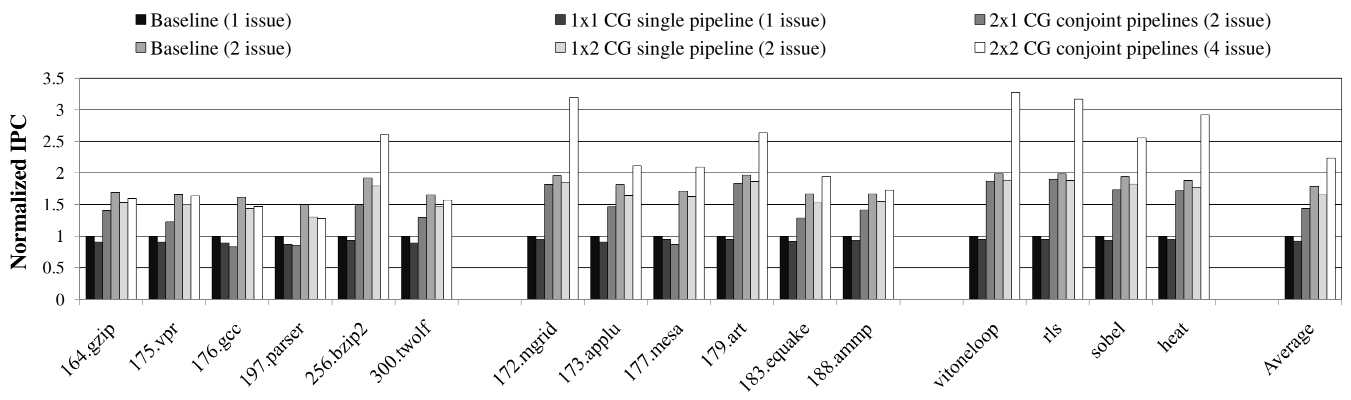


Figure 7: Single thread performance results for CG normalized to a single-issue in-order processor. The configurations are expressed as (number_of_pipelines_conjoint X issue_width_of_pipeline_stages).

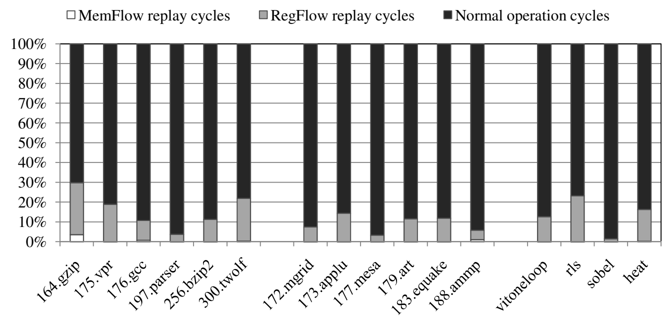


Figure 8: Contribution of memory replay cycles, register flow replay cycles and normal operation cycles to the total computational time of individual benchmarks running on a 2-issue conjoint processor. On an average, the replays contributed to about 15% of the execution time.

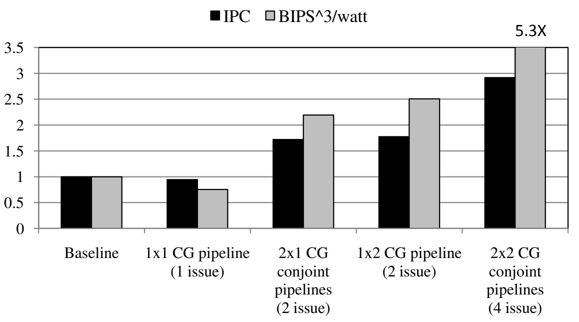


Figure 9: Comparing IPC and energy efficiency ($BIPS^3/watt$). The baseline is a single-issue in-order core (OR1200).

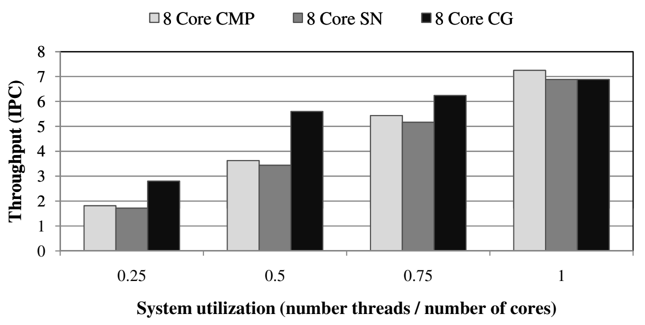


Figure 10: Throughput comparison of 8-core CMP, SN and CG systems at different levels of system utilization. A utilization of 0.5 implies that 4 working threads are assigned to the 8-core system. At this utilization, CG multicore delivers 46% throughput advantage over the baseline CMP.

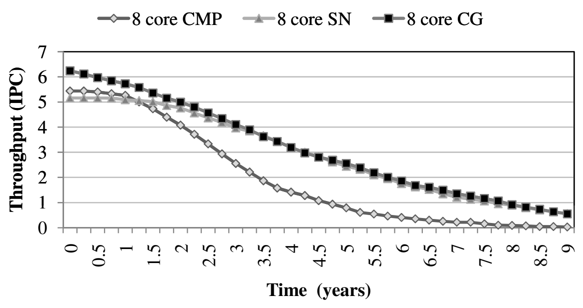
as a metric [37]. This metric is more sensitive to performance changes, and optimizing for it yields the same results as optimizing for $ED^2$ (energy times delay squared). Figure 9 shows the average IPC and $BIPS^3/watt$ comparisons for the four CG configurations normalized to the single-issue baseline processor. When going from the baseline to the single-issue CG pipeline, about 20% energy efficiency is sacrificed. However, the superior performance in wider-issue configurations, significantly improves CG's energy efficiency.

## 4.4 Multi-workload throughput

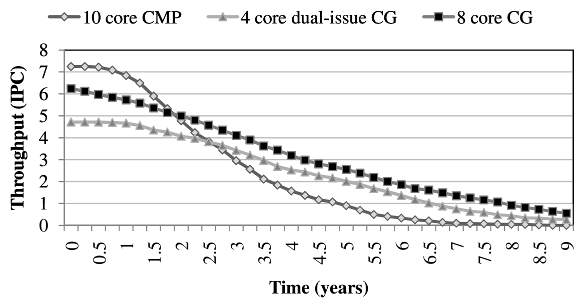
Performance of a CMP system can be measured either as the latency of thread execution (single-thread performance, prior experiment) or the rate at which jobs complete (system throughput). For the throughput comparison, three systems were compared against one another: an 8-core CMP, an 8-core SN [13] and an 8-core CG. A core here refers to a single issue in-order pipeline resource, thus an 8-core SN and CG would have eight pipelines interconnected. The system utilization was varied from 0.25 occupancy to 1.0 occupancy. This refers to the number of threads assigned to the system versus its capacity (measured as number of cores). Monte-Carlo experiments were conducted by varying the set of threads allocated to the system at each utilization level. Figure 10 shows the final throughput results from this experiment. At the peak utilization level (1.0), the 8-core CMP delivers the best throughput. This is due to the performance advantage the baseline processor has over both single pipeline SN and CG processors (see single-thread performance results above). Further, the throughput of SN and CG are identical because CG defaults to using one pipeline per thread in this peak utilization scenario. As the system utilization is lowered, CG is able to leverage the idle pipeline resources to form conjoint processors. Thus, the CG system consistently delivers the best throughput at all utilization levels $< 1$, which is a realistic expectation for over-provisioned systems.

## 4.5 Fault tolerance

The experiments so far have targeted the performance aspect of the CG architecture. In order to evaluate its reliability in the face of wearout failures, we conducted some experiments that track the throughput of the system over the course of its lifetime. For these



(a) Throughput over the lifetime of 8-core CMP, SN and CG at a fixed utilization of 0.75. After a few initial years, CG's throughput settles down to that of a SN system.



(b) Throughput over the lifetime of 10-core CMP, 4-core dual-issue CG and 8-core CG at a fixed utilization of 0.75. CG system shows the most convincing results among all the three configurations considered.

Figure 11: Lifetime reliability experiments for the various CMP, SN and CG systems. Only wearout failures were considered for this experiment.

experiments, the stages/crossbars fail as they reach their respective time-to-failures (TTFs). The system gets reconfigured over its lifetime whenever a failure is introduced. Broken stages are isolated using interconnection flexibility, and fault tolerant crossbars naturally handle crosspoint failures. A software configuration manager is re-invoked every time a failure occurs or the workload set changes. We assume a simple reconfiguration policy where: 1) all workloads are assigned a single pipeline, 2) any remaining pipelines are allocated to the threads on the basis of available ILP. The throughput of the system is computed for each new configuration based on the number of working logical pipelines and the workloads assigned to them. Monte-Carlo simulations are run for 1000 chips to get statistically significant results. The average system utilization for these experiments is kept at $0.75$. Since the throughput delivered by the CG system improves as the system utilization is lowered (see Figure 10), the CG results reported here are conservative.

Figure 11(a) shows the throughput over the lifetime for three systems: an 8-core CMP, an 8-core SN and an 8-core CG. CG clearly outperforms both of the other systems for the entire lifetime. Early on, CG achieves a throughput advantage by utilizing the idle pipelines (only 6 threads are active, leaving 2 pipelines free) to form conjoint processors. The regular CMP and SN systems cannot benefit from this. Later in the lifetime, CG sustains a throughput advantage over the CMP by effectively salvaging the working stages and maintaining a higher number of working pipelines. For instance, the CMP system's throughput drops below 2 $IPC$ around the $3.5$ year mark, whereas the CG system throughput breaches that level around the $6$ year mark. The gains add up over the lifetime, and *cumulative work done* (integral of throughput over the lifetime) advantage of CG is $68\%$ over the baseline CMP. Also note that CG's throughput converges with that of SN in the later part of the lifetime. This happens when the number of threads assigned to the system exceeds the number of working pipelines, and CG is left with no option but to default back to single pipeline processors.

Figure 11(b) compares two more system configurations to the 8-core CG: a 10-core CMP and a 4-core dual-issue CG. The 10-core CMP is chosen to have an area-neutral comparison with the CG system. The area overhead for CG is about 20% (discussed later), translating to roughly two cores for an 8-core CG system. The results show that early in the lifetime, the 10-core CMP dominates the other two configurations. This is expected as it starts off with the maximum amount of resources. However, as the failures accumulate, it quickly loses its advantage. Beyond the two year mark, the 8-core CG consistently dominates the system throughput. The 4-core dual-issue CG system performs the worst among the three. There are two reasons for this: 1) it can run fewer threads concurrently (4-cores instead of 8/10), and 2) failures in stages result in a bigger resource loss (as each stage is dual-issue).

## 4.6 Area overheads

The area for various structures that are part of the CG architecture is shown in Table 6. The overhead percentages are relative to our baseline processor - the OR1200 core. A total of five interconnection crossbars are present in the CG architecture, but since the pipelines share crossbars, its overhead is not attributable to just one pipeline. For a case where eight pipelines are connected together to form CG, each of them bears $5/8^{th}$ of the crossbar overhead. With this assumption, the total area overhead for the CG architecture is 19.6% over a traditional CMP (containing OR1200 cores).

Table 6: Area overheads from different design blocks in CG.

| **Design block** | **Area** ($mm^2$) | **Percent overhead** |
|---|---|---|
| Outstanding instructions buffer (OIB) (5 entries) | 0.037 | 5.7% |
| Store buffer (SB) (3 entries) | 0.015 | 2.3% |
| Store queue (SQ) (3 entries) | 0.021 | 3.4% |
| Bypass cache (6 entries) | 0.02 | 3.1% |
| Extra stage latches (input and output) | 0.0115 | 1.8% |
| Miscellaneous logic | 0.055 | 0.9% |
| 8x8 fault tolerant crossbar (with interconnection wires) five such crossbars are shared between eight pipelines | 0.025 | 3.9% |
| Total area overhead of CG | | **19.6%** |

## 4.7 Power overheads

The power overhead in CG comes from three sources: crossbars, stage/crossbar interconnection and miscellaneous logic (extra latches, new modules). Table 7 shows the breakdown, with total power overhead at $16.9\%$. The actual power numbers in the table are overheads for one CG pipeline while it is part of a 2-issue CG conjoint processor. Note that a part of this overhead will be there even in a traditional 2-way superscalar (relative to having 2 independent 1-way pipelines).

Table 7: Power overhead for CG. These overheads are reported with OR1200 power consumption as the baseline.

| **Component** | **Power overhead pipeline** ($mW$) | **Percent overhead Percent overhead** |
|---|---|---|
| Crossbars | 4.0 | 4.26% |
| Interconnection links | 5.8 | 6.19% |
| Other design blocks | 6.1 | 6.38% |
| Total power overhead | | **16.9%** |

## 5. CONCLUSION

In the multicore era, where on one hand abundant throughput capabilities are being incorporated on die, single-thread performance and power efficiency challenges still confront the designers. Further, the increasing process variation and thermal densities are stressing the limits of CMOS scaling. To efficiently address all these solutions, designers can no longer rely on an evolutionary design process. Further, simply combining existing research solutions for performance and reliability is neither easy nor cost-effective. In this paper, we presented CoreGenesis, a highly adaptive multiprocessor fabric that was designed with performance and reliability targets from the ground up. The interconnection flexibility within CoreGenesis not only ensures impressive fault-tolerance, but coupled with the addition of decentralized instruction flow management, it can also merge pipeline resources to accommodate dynamically changing application requirements. Our experiments demonstrate that merging of two pipelines within CoreGenesis can deliver on average 1.5X IPC gain with respect to a standalone pipeline. In a CMP, with only half of its cores occupied, this merging can enhance throughput performance by 46%. Finally, the lifetime reliability experiments show that an 8-core CoreGenesis chip increases the cumulative work done by 68% over a traditional 8-core CMP.

## 6. ACKNOWLEDGMENTS

We thank the anonymous referees for their valuable comments and suggestions. The authors acknowledge the support of the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This research was also supported by National Science Foundation grants CCF-0916689 and ARM Limited.

## 7. REFERENCES

[1] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar. 2005.

[2] L. Seiler *et al.*, "Larrabee: a many-core x86 architecture for visual computing," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 1–15, 2008.

[3] Tilera, "Tile64 processor - product brief," 2008, http://www.tilera.com/pdf/.

[4] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," in *Proc. of the 36th Annual International Symposium on Microarchitecture*, Dec. 2003, pp. 81–92.

[5] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[6] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The impact of technology scaling on lifetime reliability," in *Proc. of the 2004 International Conference on Dependable Systems and Networks*, Jun. 2004, pp. 177–186.

[7] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 1, pp. 33–38, 2008.

[8] E. Ipek, M. Kirman, N. Kirman, and J. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *Proc. of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 186–197.

[9] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," in *Proc. of the 40th Annual International Symposium on Microarchitecture*, Dec. 2007, pp. 381–393.

[10] D. Tarjan, M. Boyer, and K. Skadron, "Federation: Repurposing scalar cores for out-of-order instruction issue," in *Proc. of the 45th Design Automation Conference*, Jun. 2008.

[11] H. H. Najaf-abadi and E. Rotenberg, "Architectural contesting," in *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, 2009, pp. 189–200.

[12] B. F. Romanescu and D. J. Sorin, "Core cannibalization architecture: Improving lifetime chip performance for multicore processor in the presence of hard faults," in *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[13] S. Gupta, S. Feng, A. Ansari, J. A. Blome, and S. Mahlke, "The stagenet fabric for constructing resilient multicore systems," in *Proc. of the 41st Annual International Symposium on Microarchitecture*, 2008, pp. 141–151.

[14] P. Salverda and C. Zilles, "Fundamental performance constraints in horizontal fusion of in-order cores," in *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, Feb. 2008.

[15] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proc. of the 22nd Annual International Symposium on Computer Architecture*, Jun. 1995, pp. 414–425.

[16] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "The multicluster architecture: Reducing cycle time through partitioning," in *Proc. of the 30th Annual International Symposium on Microarchitecture*, Dec. 1997, pp. 149–159.

[17] J. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: MIT Press, 1985.

[18] A. Baniasadi and A. Moshovos, "Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors," in *Proc. of the 33rd Annual International Symposium on Microarchitecture*, 2000, pp. 337–347.

[19] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 87–96, 2004.

[20] D. Sylvester, D. Blaauw, and E. Karl, "Elastic: An adaptive self-healing architecture for unpredictable silicon," *IEEE Journal of Design and Test*, vol. 23, no. 6, pp. 484–490, 2006.

[21] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," in *Proc. of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 470–481.

[22] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *Proc. of the 36th Annual International Symposium on Computer Architecture*, Jun. 2009.

[23] ARM, "Arm9," http://www.arm.com/products/CPUs/families/ARM9Family.html.

[24] A. Ansari, S. Gupta, S. Feng, and S. Mahlke, "Zerehcache: Armoring cache architectures in high defect density technologies," in *Proc. of the 42nd Annual International Symposium on Microarchitecture*, 2009, pp. 100–110.

[25] Alpha, "21364 family," 2001, http://www.alphaprocessors.com/21364.htm.

[26] S. Satpathy, Z. Foo, B. Giridhar, D. Sylvester, T. Mudge, and D. Blaauw, "A 1.07 tbit/s 128128 swizzle network for simd processors," in *Proc. of the 2010Symposium on VLSI Technology*, 2010.

[27] S. Gupta, A. Ansari, S. Feng, and S. Mahlke, "Stageweb: Interweaving pipeline stages into a wearout and variation tolerant cmp fabric," in *Proc. of the 2010 International Conference on Dependable Systems and Networks*, Jun. 2010.

[28] T. T. Ye, L. Benini, and G. D. Micheli, "Analysis of power consumption on switch fabrics in network routers," in *Proc. of the 39th Design Automation Conference*, 2002, pp. 524–529.

[29] Trimaran, "An infrastructure for research in ILP," 2000, http://www.trimaran.org/.

[30] OpenCores, "OpenRISC 1200," 2006, http://www.opencores.org/projects.cgi/web/ or1k/openrisc_1200.

[31] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August, "The liberty simulation environment: A deliberate approach to high-level system modeling," *ACM Transactions on Computer Systems*, vol. 24, no. 3, pp. 211–249, 2006.

[32] R. Kumar, N. Jouppi, and D. Tullsen, "Conjoined-core chip multiprocessing," in *Proc. of the 37th Annual International Symposium on Microarchitecture*, 2004, pp. 195–206.

[33] ITRS, "International technology roadmap for semiconductors 2008," 2008, http://www.itrs.net/.

[34] PTM, "Predictive technology model," http://ptm.asu.edu/.

[35] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, and S. Ghosh, "Hotspot: A compact thermal modeling method for cmos vlsi systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 5, pp. 501–513, May 2006.

[36] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," in *Proc. of the 31st Annual International Symposium on Computer Architecture*, Jun. 2004, pp. 276–287.

[37] D. Brooks, V. Tiwari, and M. Martonosi, "A framework for architectural-level power analysis and optimizations," in *Proc. of the 27th Annual International Symposium on Computer Architecture*, Jun. 2000, pp. 83–94.