Tartan: Evaluating Spatial Computation for Whole Program Execution

Mahim Mishra,[†] Timothy J. Callahan,[†] Tiberiu Chelcea,[†] Girish Venkataramani,[†] Mihai Budiu[‡] and Seth C. Goldstein[†]

[†]Carnegie Mellon University
Pittsburgh, PA
{mahim,tcal,tibi,girish,seth}@cs.cmu.edu

[‡] Microsoft Research
Mountain View, CA
mbudiu@microsoft.com

Abstract

Spatial Computing (SC) has been shown to be an energy-efficient model for implementing program kernels. In this paper we explore the feasibility of using SC for more than small kernels. To this end, we evaluate the performance and energy efficiency of entire applications on Tartan, a general-purpose architecture which integrates a reconfigurable fabric (RF) with a superscalar core. Our compiler automatically partitions and compiles an application into an instruction stream for the core and a configuration for the RF. We use a detailed simulator to capture both timing and energy numbers for all parts of the system.

Our results indicate that a hierarchical RF architecture, designed around a scalable interconnect, is instrumental in harnessing the benefits of spatial computation. The interconnect uses static configuration and routing at the lower levels and a packet-switched, dynamically-routed network at the top level. Tartan is most energy-efficient when almost all of the application is mapped to the RF, indicating the need for the RF to support most general-purpose programming constructs. Our initial investigation reveals that such a system can provide, on average, an order of magnitude improvement in energy-delay compared to an aggressive superscalar core on single-threaded workloads.

Categories and Subject Descriptors C.1.3 [*Processor Architectures*]: Data-flow architectures, Hybrid systems; B.6.3 [*Design Aids*]: Automatic synthesis, Simulation; B.8.1 [*Performance and Reliability*]: Reliability, Testing and Fault-Tolerance; D.3.4 [*Processors*]: Compilers

General Terms Design, Performance

Keywords spatial computation, dataflow machine, reconfigurable hardware, asynchronous circuits, low power, defect tolerance.

1. Introduction

Market demand and technology push, in the guise of thermal dissipation, energy-density, complexity constraints and wire-delay, are changing the equation that has, until recently, made complex out-of-order superscalar designs the workhorse of computing. No longer can architects rely on increasing the clock speed and complexity of the pipeline as the major means of improving performance. In fact, demand for low power devices may mean that performance may no longer be the crucial metric of success. This changing landscape has prompted many to explore alternative architectures that harness coarse-grained parallelism; for example, commercial multi-core chips capitalize on parallelism by replicating a number of simple cores, thereby boosting scalability. While these chips are focused on server workloads which are rich in threads, it is not yet clear how single-threaded applications can benefit from such architectures. The research community has responded with tile-based architectures [30, 37, 38] that harness coarse-grained parallelism on an array of light-weight processors. In this paper, we explore an alternative array-based architecture that uses spatial computation (SC) to improve energy efficiency as measured by the energy-delay product.

Spatial computation (SC) is a model of computation optimized for wires: it lays out the computation in space rather than in time. In earlier work [8], we presented one instantiation of SC: a compilation framework that can synthesize ANSI-C programs into gate-level asynchronous circuit descriptions, which are then laid out as ASICs using commercial tools. The previous evaluation was restricted to selected media processing kernels, on which we demonstrated that the synthesized ASIC was better than an aggressive 4-wide superscalar by up to three orders of magnitude in energy efficiency, while delivering comparable performance. In Garp [9], speedups of up to 10x are achieved when mapping small kernels compiled from C to a reconfigurable fabric. These results raise the question: can the SC model be used to form the basis of a scalable general-purpose processing system for whole program execution?

To answer this question we explore a hybrid architecture, Tartan, which integrates a large reconfigurable fabric (RF) with a simple processor core. Tartan is designed to be:

- General Purpose: The system must be able to execute a wide variety of programs written in standard high-level languages, e.g., ANSI-C. Applications are compiled into an instruction stream and an RF configuration. The instruction stream is executed on Tartan's processor core, which is more efficient for control intensive tasks such as the OS. The configuration describes an SC circuit and is loaded onto the RF, which is more efficient for computation intensive tasks.
- Energy Efficient: Tartan uses its RF to reduce energy consumption—an RF configuration describes an application-specific dataflow machine, thereby saving on the power consumed by instruction fetch and decode in (processor) grid architectures. Further, the RF is implemented as a fully distributed asynchronous circuit which has been shown to be energy efficient [8].
- Fabrication Friendly: Future technologies will place stringent demands on manufacturing. Tartan's RF reduces these demands by tolerating manufacturing defects and both intra- and inter-die parametric variation. Tartan implements an extreme version

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

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.
Copyright © 2006 ACM 1-59593-451-0/06/0010. . . $5.00.

of SC on the RF—computation structures are never shared, and each program operation is implemented on a dedicated functional unit. This results in circuits which use only local communication, require neither broadcast nor global control, and are self-timed. The distributed nature of the SC circuits combined with the asynchronous RF enables placement and routing around defects [17]. While this strategy affects local timing around the defective area, the absence of a global clock ensures that timing closure is not an issue. Furthermore, many styles of self-timed circuits are naturally tolerant of parametric variation [14]. Finally, the regular layout of the RF eliminates many of the issues that arise in deep-submicron lithography and fabrication.

Tartan's many advantages arise directly from the extreme form of SC that it implements. However, this comes at a cost: as more of the program is placed on the RF, the distance (and thus latency) between two functions on the RF and between a function on the RF and shared memory increases. The heart of this paper (Section 5) is an exploration of the tradeoffs involved in resolving this fundamental tension in SC when implementing large programs on an RF. In particular, we examine the cost of communicating across the fabric, the cost of maintaining memory coherence, the effect of varying page sizes, and the impact of defect tolerance. These trade-offs are evaluated in terms of area usage, execution time, power, and most importantly, energy-delay. We use a timing and energy accurate simulator (described in Section 4) of both the CPU core and the RF to carry out our experiments. Our simulator allows us to not only determine the overall performance of a design, but to also quantify the time and energy spent in various circuit components.

Sections 2 and 3 describe the Tartan architecture and compilation methodology. Section 6 covers related work. In Section 7 we conclude. Our results show that a hybrid architecture such as Tartan is capable of harnessing—for whole programs—the energy and power benefits shown for kernels implemented as ASICs [8]. Tartan is up to an order of magnitude more energy efficient than an aggressive superscalar core. We found, however, that to harness this energy efficiency, it is important to execute as much of the program on the RF as possible. To achieve this, it is essential for the RF to directly implement most common programming constructs (such as procedure calls) and to act as a peer of the CPU core, invoking services from it. This also allows Tartan to use a simple processor core instead of an aggressive, power-hungry superscalar core. Finally, our results indicate that supporting large programs with good performance requires distributing memory on the fabric.

2. System Architecture

Tartan is a hybrid architecture consisting of a RISC CPU core and a hierarchical, medium-grained, clockless reconfigurable fabric (RF). Figure 1a shows a high-level view of the system. The user application is partitioned into two parts, one executing on the CPU and the other on the RF. The CPU and RF interact as peers in this system [7]: each part invokes services from the other through automatically-generated stubs, much like Remote Procedure Calls [26]. Both compute engines share the same memory space; this can be accessed through a shared L1 data cache (shown in Figure 1a and evaluated in this paper), or through separate, hardware-coherent L1 caches. Our design currently has only a single thread of control, which may be active on the CPU or RF. However, in principle, there could be multiple threads both on the CPU and RF, with both parts of the design active simultaneously.

We first describe the CPU and required ISA extensions for communicating with the RF, followed by a description of the RF.

2.1 CPU Core

The CPU runs the operating system, bootstraps user applications, and executes control-intensive parts of the application that do not

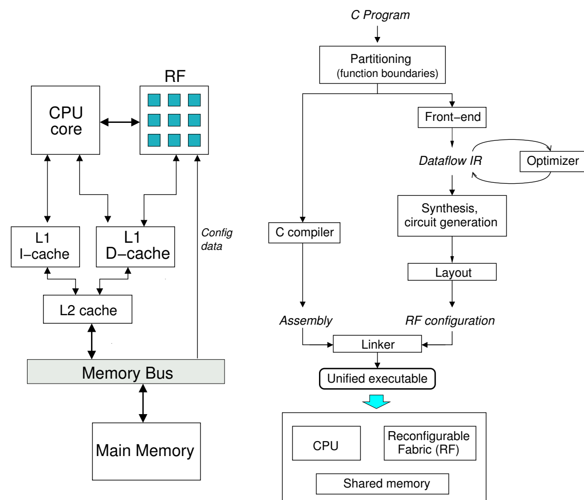

Figure 1. *(a) Tartan Architecture (b) Compiler Flow*

INITRFU k:	initialize RF routine k.
IPRFUI R1,R2,R3:	send three integer register values to the RF.
IPRFUF F1,F2,F3:	send three fp register values to the RF.
STRFU k:	start the execution of RF routine with ID k.
OPRFUI R1,R2:	read into integer registers up to two values from the RF.
OPRFUF F1,F2:	read into fp registers up to two values from the RF.
CONTRFU R1:	block the core until the RF raises an interrupt; on wakeup, copy one value from the RF bus register `R1`.

Table 1. *New instructions in the CPU core to interact with the RF*

map efficiently to the RF. The CPU can also be used to run library routines and legacy applications which, in the absence of source code, cannot be recompiled to run on the RF. In principle, the CPU core can be anything which fits the application requirements and the transistor and power budget. If a significant portion of the application is expected to run on the CPU, it can be a superscalar, out-of-order core. However, if the major portion of execution time is expected to be spent on the RF, then a narrow-issue in-order core is appropriate. In this paper, we evaluate both a wide-issue out-of-order core as well as a narrow in-order core.

2.2 CPU-RF Interface

The CPU and RF are connected by a wide bus to transfer data and control information. The bus is as wide as the number of read ports into the CPU's register file; we model a 96-bit bus for 3 read ports. There are also a small number of control lines for instruction identification and interrupt signaling.

We augment the core ISA with instructions from [7], as shown in Table 1. The `INITRFU` and `STRFU` instructions put the RF function identifier on the data bus and set the appropriate control lines. If the core does not have any useful computations to perform after invoking the RF, it can execute the `CONTRFU` instruction. This blocks the core until an interrupt is received from the RF indicating that it has finished computation or wants to call something on the core. If the RF computation is expected to be of long duration, the core can shut down (by clock-gating or stopping its clock entirely) to save power. It may also be possible to implement many sleep modes on the core, each of which will save progressively more power and will take longer to wake up from on an RF interrupt. An `IPRFU` instruction transfers up to 3 data values from the register file to the data bus; an `OPRFU` instruction transfers up to 2 values from the bus to the register file.

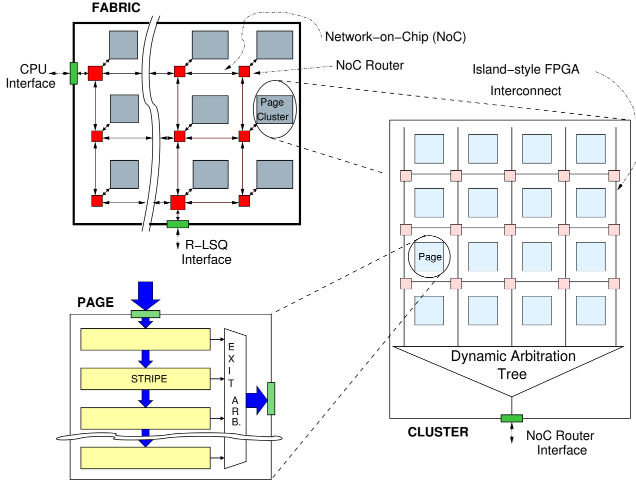

Figure 2. *The hierarchical RF architecture*

We use mixed-timing FIFOs (based on [11]) to interface between the synchronous CPU and clockless RF. When sending data from the CPU to the RF, the RF can read data from the FIFOs whenever available, without synchronization overheads. When sending data from the RF to CPU, the FIFOs synchronize the asynchronous packets to the CPU's clock, with an added latency of one cycle.

2.3 RF architecture

The Tartan RF, as shown in Figure 2, is hierarchical. At the lowest level (bottom-left image), a small number of stacked *stripes* of processing elements constitute an RF *page*. A small array of pages are grouped into a *cluster* (right image). Finally, clusters are stitched together into a fabric by the Network-on-Chip (NoC). Each level has a different interconnect architecture specialized to the amount and pattern of communication expected at that level. Each element of the computation and interconnect architecture is designed using self-synchronized, clockless circuits. We next describe each level of the design hierarchy.

RF Page. An RF page resembles the Piperench architecture [15]. It consists of a vertical stack of 16 stripes, and each stripe consists of a set of medium-grained (ALU-based as opposed to LUT-based) processing elements (PEs). We follow [15] in using 16 PEs per stripe with a default PE width of 8 bitops, adding up to 128 bitops per stripe. Tartan makes extensive use of predicated execution (as described in Section 3); to support this, we also add some single-bit PEs to the stripes. ALU operations of wider bitwidths can be performed by connecting multiple PEs together. Some operations, e.g., integer multiply, may require multiple stripes, and the architecture naturally supports pipelined multiplication. It is not necessary for all the stripes to be homogeneous; we envisage a mix of stripes where some have PEs of wider granularities or specialized implementations of operations. We are in the process of identifying a mix that would be a good fit for a wide range of applications.

Data always enters at the top of the page, and flows down the page through stripes of computation. There are no feedback connections within a page and the stripe-to-stripe interconnect is a partial crossbar. The page is designed to be a natural fit for the predicated hyperblocks generated by our compiler (see Section 3) and each hyperblock is mapped to one or more pages.

There are two exceptions to the straight-line computation rule—memory accesses and procedure call operations. These split-phase operations use the 16-way arbitration element in each page to access the cluster's dynamically arbitrated tree, which provides access to the cluster's Network-on-Chip (NoC) router.

Cluster level. A group of 4x4 pages form a cluster. The cluster architecture is conceptually similar to an island-style FPGA but with much coarser granularity, i.e., the logic block is a page instead of a look-up table (LUT). The cluster interconnect consists of channels communicating through switch-boxes, and its configuration is statically determined by the compiler. The only exception is the dynamic arbitration tree (based on [43]), which provides pipelined access to the communication port between the cluster and the NoC.

Chip level. At the chip level, the clusters are connected together by a dynamically routed, packet-switched NoC. This is ideally suited to the sporadic, statically unpredictable communication (procedure calls, returns, and memory accesses) that occurs at the global level. The network has one router (based on [2]) per cluster.

The interconnect is designed to exploit the difference in locality of communication between the stripes, pages, and clusters. The first two levels are statically configured, provide high bandwidth and low latency and consume less energy, which is a good match for the predictable, dense and frequent communication within a hyperblock or procedure. At the global level, communication is more sporadic and unpredictable, making a dynamically routed NoC a more resource-effective choice.

3. Compilation

Our compiler flow is shown in Figure 1b. A program is partitioned for execution on the core and RF. After partitioning, each part is compiled by a separate toolflow and linked together at the end.

CPU-RF partitioning. Our compiler splits applications into CPU and RF components at procedure boundaries. A procedure on the RF can call other procedures on the RF as well as procedures on the core. Calling across the CPU-RF interface is achieved by automatically generated software stubs on the CPU-side and by stub-like hardware on the RF that directs operand values and procedure identifiers to the CPU through the CPU-RF FIFOs.

As mentioned in Section 2.1, we expect OS routines, control-intensive application code and legacy applications and libraries to execute on the CPU. It is also possible to implement two copies of a piece of code, one each on the CPU and RF, and invoke one or the other based on caller location and predicted execution time. This can be especially beneficial for a number of C library routines like *memcpy*. In our current partitioning, we implement as much of the application as we can on the RF to optimize for energy-delay (since the RF always consumes much less energy).

Translating C functions to circuits. The translation of a C function to a spatial RF mapping occurs in three steps—(1) compiling C to a predicated, dataflow intermediate representation (IR) called Pegasus [5, 6]; (2) synthesizing the IR to a logic netlist; and (3) placing and routing the netlist on the RF. Pegasus is based on Predicated Static Single Assignment (PSSA) [12, 10] and is extended to handle memory dependences in a unified manner. A function is represented by a directed graph in which nodes are operations and edges indicate value flow, including the special case of *token* edges that are inserted to enforce the dynamic execution ordering of two nodes [1], e.g., between a store and a load.

The basic unit of execution is the hyperblock [24]. Control flow within each hyperblock is converted to data flow using predicates—boolean data values that encode which control path would have been taken in the original control flow graph. All safe operations from all paths are performed unconditionally; at merge points, lenient multiplexors controlled by predicates are inserted to select the correct values to use in subsequent computation (these muxes correspond roughly to SSA ϕ–nodes). Only operations with side effects such as stores and calls require a predicate input to conditionally suppress their execution. Predicates also steer the final values produced by a hyperblock to the inputs of the correct succes-

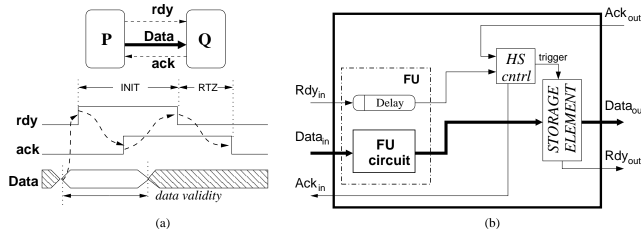

Figure 3. *(a) Pipeline communication is achieved through the asynchronous, localized bundled data handshake protocol; (b) the architecture of a clockless data-triggered pipeline stage*

sor hyperblock—i.e., they implement a distributed, dataflow version of a conditional branch. Each final value is sent as soon as it and its guiding predicate are known, so that computation in the successive hyperblock(s) can begin while computation in the current hyperblock is still finishing. In the case of a loop, this allows self-organizing software pipelining.

The compiler instantiates each IR node as a clockless data-triggered pipeline stage with a function unit (FU), an output register and a handshake controller (HS Cntrl) as in Figure 3. The FU starts its computation as soon as all its inputs are available and latches the result as soon as its output register is empty (i.e., the previous output has been consumed). Each IR data edge translates to a channel between the corresponding units. Each channel consists of a data bus plus control signals for asynchronous 4-phase bundled-data handshaking [36]; an IR token edge becomes a bundle with just the control signals. This approach localizes all control constructs and eliminates the global clock and energy wasted on its distribution and also alleviates timing closure problems [42].

Memory. The IR has LOAD and STORE nodes for accessing memory. The corresponding node implementations in an RF page connect to a 16-way arbitration element for access to the cluster-wide dynamically-arbitrated tree interconnect, which in turn connects to the NoC and thence to the memory subsystem (see Section 2.3). Each memory access node has a predicate input. A LOAD may execute speculatively, where a request is sent to memory before the predicate's value is known; if the predicate turns out to be false, any exception the LOAD may have caused is ignored.

The compiler adds token edges to explicitly synchronize operations which share an ordering dependency. For example, if a LOAD occurs after a STORE in program order, and if alias analysis cannot disambiguate these accesses as being independent of each other, then a token edge connects the STORE to the LOAD. Tokens encode true-, output- and anti-dependences, and are "may" dependences.

Procedures. Tartan and its toolflow directly support actual procedure calls (not just compile-time inlining). The caller sends to the callee the arguments as well as (i) a pointer to the continuation point, (ii) the current stack pointer, and (iii) a memory token, which enforces ordering between memory accesses in the caller before the call point and accesses in the callee. The callee can be located on either the RF or the CPU; this flexibility is important since a typical procedure may call a library function, which in our current implementation is performed on the CPU. Potentially recursive calls save all local values to the stack before the call and restore them after.

For function pointers on the RF, a simple lookup mechanism resolves the pointer either to a location on the RF or to a procedure in the CPU-resident code; with our pointer encoding and resolution we are able to automatically handle function pointers on the CPU and RF that may point at procedures implemented on either side.

Mapping circuits to the RF. If Tartan is to be used for general-purpose computing, the circuits generated by the compiler need to be mapped to the RF in time comparable to software compilation rather than hardware design. This is possible because the fabric architecture has the following properties:

- The logic resources have a relatively coarse granularity (e.g., 8-bit adders instead of LUTs), reducing the number of placeables.
- The hardware resources closely match the high-level structure of the circuit: a feed-forward hyperblock naturally maps to one or more pages, and one or more clusters can accommodate a procedure. This allows for high-level information about the circuits to be preserved and utilized.
- The RF includes handshake and arbitration elements to allow easy implementation of asynchronous circuits, so that the timing closure problem is eliminated.

4. Evaluation Methodology

Our simulator infrastructure, *SpatialSim*, integrates a CPU simulator with an RF simulator. For our evaluation, we used benchmarks from the Mediabench [23], SPECINT 1995 [34] (*go*, *m88ksim*, *compress*, *li*, *ijpeg*, *perl*) and SPECINT 2000 [35] (*vpr*, *mcf*, *twolf*) suites. Unless otherwise noted, we mapped the whole program onto the RF, leaving only the C library and a small number of other functions (e.g., those that called *longjmp*, since we have not yet implemented a means of unwinding the RF stack and rolling back execution state) on the CPU. We simulated every benchmark to completion, without employing fast-forwarding or functional simulation at any point. Where multiple input sets were available, we chose the smallest input. These choices were made for the following reasons:

1. Our compilation and simulation frameworks are still under active development, and some of the omitted benchmarks stretch the capabilities of our simulation framework. For the benchmarks that we do simulate, partitioning and compilation were performed completely automatically by our toolflow.
2. The RF simulator performs a very detailed event-based simulation at a granularity similar to RTL simulation, and is over 20x slower than cycle-accurate CPU simulators on the same code. For the larger benchmarks, simulation time was prohibitively large ($>$ 48 hours) even for the smallest input sets.

4.1 CPU simulation

The CPU simulator is an extension of the cycle-accurate MASE simulator [22] that supports implementations of the RF instructions as described in Section 2.2. The CONTRFU instruction has a latency of 5 cycles; all other RF instructions have single-cycle latencies. The latency of CPU-RF communication is based on timing models derived from [11].

To model energy consumption on the CPU, we ported the Wattch energy models [3] to SpatialSim. We use Wattch's *cc3* clocking scheme, i.e., all active units consume full power while all non-active units consume 10% of full power, which corresponds to a state-of-the-art, aggressively clock-gated processor. When the CPU is blocked on a CONTRFU instruction, we assume that it consumes negligible power. The 5 cycle CONTRFU latency is spent in powering-up the processor. This is conservative compared to modern processors which support low-power inactive modes with single-cycle wakeup [19].

For the Tartan processor, we model two different cores (see Table 2). The latency and power estimates we use for the RF simulation are based on an implementation of the RF circuits in a 180nm process; CPU1 is chosen to be an aggressive superscalar processor from the same technology generation.[1] CPU2 has lower performance and consumes less power, and is better suited for

[1] Higher frequencies can be achieved with the 180nm process; however, since our circuit implementations are also far from optimal (in particular due to a lack of asynchronous gates in the standard cell library), we believe our choice is fair.

	CPU1	CPU2
Clock	667 MHz	667 MHz
Issue Width	4	2
Issue Order	Out-of-order	In-order
Fetch Q	16	8
ROB Size	16	8
LSQ Size	8	8
ALUs	4 Int, 4 FP	2 Int, 1 FP
Branch Penalty	6 cycles	6 cycles
L2 miss latency	18 cycles	18 cycles
Mem Bus	8 B	8 B
L1 D-cache	8K, 4way, 32B, 1 cyc	8K, 4way, 32B, 1 cyc
L1 I-cache	8K, direct, 32B, 1 cyc	4K, direct, 32B, 1 cyc
L2 cache	256K, 4way, 64B, 6 cyc	256K, 4way, 64B, 6 cyc

Table 2. *CPU core parameters*

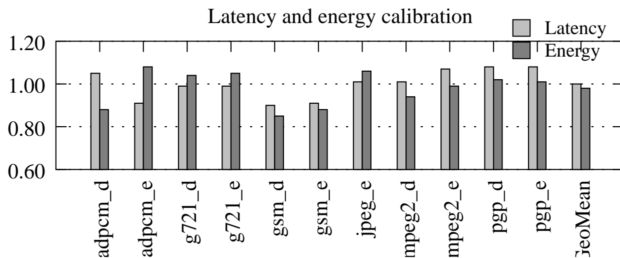

Figure 4. *Accuracy of the timing and energy estimation models. Each bar represents the ratio of the value measured in* SpatialSim *to ASIC estimation for program kernels used in [8] (note that the Y-axis starts at 0.6).*

an application where the RF is expected to do most of the work. When comparing Tartan against a superscalar core, we use CPU1 as the baseline, since it has better performance and energy-delay (but worse energy) than CPU2.

It should be noted that our power numbers exclude the power consumed in the L1 and L2 caches and TLBs (recall that in our current evaluation the RF shares the L1 and L2 caches with the CPU). We also do not model the power consumed in I/O buses, processor pins etc. This study therefore demonstrates the energy-efficiency benefits obtainable for the execution core. For domains such as embedded systems which have very limited memory and I/O subsystems, these results are a very good indication of the energy-efficiency benefits that can result for the entire system.

4.2 RF simulation

SpatialSim uses an event-driven simulation to accurately simulate the asynchronous circuits generated by the compiler. The latency and energy estimations for the RF units were derived from implementations of the circuits in a [180nm/2V] technology (based on [42]). The latency and energy consumption of a computation node depend on the bitwidth, number of inputs, number and value of constant-inputs, and output fanout. We extracted latency and energy models for each type of computation node and for each of these parameters using industrial CAD tools; during simulation, these models are used to estimate the global power and latency of a given application.

Communication latencies modeled in *SpatialSim* are determined as follows:

- Intra-page communication is accounted for by incorporating the average capacitative load extracted from layout into stripe latency computation.
- Inter-page communication within a cluster depends on the number of hops. We model a per-hop latency of 80ps, which is equivalent to one tri-state buffer driving twice the average load extracted from circuit layout.
- NoC communication also depends on the number of hops. Based on estimates from [2], we model a per-hop latency of 1ns.
- We model contention in the arbitration trees in the page clusters using models from [41].

We have implemented a greedy placement algorithm that uses an application's call and memory access profiles to determine the connection weights between different hyperblocks and between hyperblocks and memory ports. Each hyperblock is then assigned to one or more pages on the RF, keeping hyperblocks with high connection weights close to each other, and hyperblocks with a large number of memory accesses close to memory.

Simulation accuracy. To measure the accuracy of our delay and energy models, we synthesized a number of kernels from the MediaBench suite (the same kernels as used in [8]) down to Verilog, and performed layout using industrial CAD tools. We compared the latency and energy obtained from post-layout simulations with the energy and latency reported by our simulator. Figure 4 shows that our latency estimates are within 10% from the Verilog numbers, our energy estimates are within 15%, while the geometric means are 1% and 2.5% off, respectively. We use the GM as a measure for the error (and not, for example, an average of the absolute errors) because our method underestimates the latencies and energies for some kernels while it overestimates some others, and these errors can be expected to cancel out for a large program.

Simulation limitations. The results reported in this paper can be considered a limit study, because of the following simplifications and approximations in our simulation:

- We assume that the RF is large enough to accommodate the entire program without requiring any reconfiguration and virtualization during the course of the execution (we address a more limited RF in Section 5.6).
- We do not include the time and energy of reconfiguration in this study.
- We do not perform any routing currently, but assume that enough routing resources are available to allow us to always take the shortest path (based on hopcount) through the static interconnect.
- The latency and energy estimates used for compute units are derived from an ASIC implementation. A reconfigurable implementation could be slower because of muxes at the output of each PE, and because of extra loads on wires. In [21], FPGAs are measured to be 2-4x slower than ASICs on a range of applications. However, the inaccuracy in our simulations is likely to be much smaller, because a) most of the slowdown and extra energy consumption in an FPGA is attributable to routing rather than logic (as shown in [32]), and the latency and energy of the interconnect in our simulations are modeled closely after a reconfigurable implementation; and b) we use a coarse-grained PE instead of the single-bit LUTs used in [21], which have a much smaller slowdown (e.g., [21] indicates that the slowdown on an FPGA is 2x or less when the application can use coarse-grained on-chip resources such as DSPs in addition to single-bit LUTs). To bound the performance and energy-delay degradation, we also report on experiments with slower compute nodes (see Section 5.4).
- We model contention in the memory access trees, but not in the chip-level NoC. A newer version of SpatialSim that more accurately models the NoC shows that congestion has less than 5% effect on performance. This is expected, as most of the contention in the RF is due to concurrent memory accesses which are correctly accounted for in the memory trees, leaving little or no contention in the NoC.

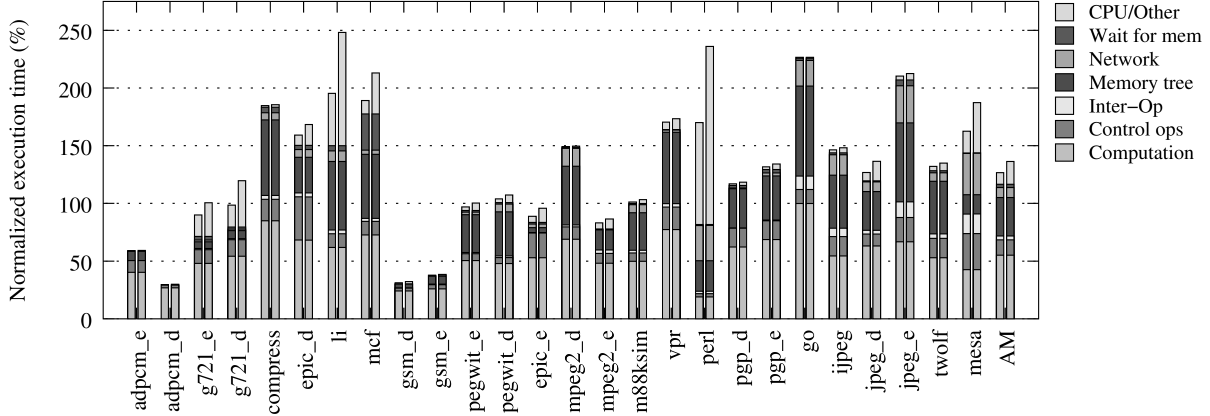

Figure 5. *Breakdown of execution time on the RF, normalized to the program running alone on* `CPU1`*. The bars on the left are for* `OPT`*, while the bars on the right are for* `NARROW`*.* AM *is the Arithmetic Mean.*

- We simulate per-procedure memory access trees instead of per-cluster trees, often resulting in deeper arbitration structures. Our initial investigation indicates that this assumption increases the delay in our current simulations by up to 35% on some benchmarks.
- We model each page as a set of bit-operations that can be configured into arbitrary operations using an appropriate number of bitops (a 1-bit predicate operation takes 1 bit-op, a 32-bit add takes 32 etc.).
- Our energy measurements do not account for static power. While this is not a concern at the 180nm node, it will be at 90nm and beyond, especially with the large RF size we propose. Many circuit, logic–design and device techniques have been proposed to reduce leakage power (see [20] for a survey); we are exploring the applicability of these to Tartan.

5. Evaluation Results

As is obvious, Tartan uses significantly more hardware resources than the CPU alone. In this evaluation, we show how well Tartan's architecture and compilation framework use these extra resources to improve energy–delay, not just performance. We use the `CPU1` from Table 2 as our baseline. The results we present measure latency and energy-delay for a number of benchmarks running on `CPU1` alone and on Tartan configured with a CPU matching CPU1's configuration (henceforth referred to as `OPT`), and one configured with CPU2 (henceforth called `NARROW`). We also quantify the effects of changing various parameters associated with our design.

5.1 Performance

We configured SpatialSim to perform the most realistic simulation of Tartan, generating placements for the circuits, accounting for communication costs at all levels of the hierarchy, and using a realistic cache hierarchy. We ran these experiments for both `OPT` and `NARROW` configurations. Figure 5 presents the performance results for this simulation normalized to the `CPU1` execution time (a bar of height 200 means `OPT` was twice as slow as `CPU1` alone). At any point in the execution, a large number of events may be occurring in the RF at once, including computation, communication and memory accesses. We want to see how much of the latency of the non-compute operations can be hidden by computation and which non-compute operations form bottlenecks in the system when there is no useful work going on. For this purpose, we segment execution time as follows: at any simulation tick, if any compute node is active in the RF, that tick is attributed to "Computation", even if other type of nodes process events; when no compute node is active, if any control node (such as a mux) is active, that tick is attributed to "Control Ops"; when neither compute nor control nodes are active, if any intra-cluster communication is happening, that tick is attributed to "Inter-Op"; and so on successively for "Memory Tree" (when the intra-cluster dynamically-arbitrated memory access tree is active), "Network" (when the inter-cluster NoC is active), "Wait for mem" (when the RF is blocked waiting for a memory access to complete) and "CPU/Other" (when either the CPU is computing, or a CPU–RF communication is in progress). The benchmarks are sorted from left to right in the order of increasing circuit size (i.e., *adpcm_d* is the smallest while *mesa* is the largest). The circuit size is closely related to benchmark code size, but is not directly proportional due to variable amounts of loop unrolling in different loops.

The graph in Figure 5 shows that the average slowdown for `OPT` is 25%, while that for `NARROW` is 30%. For `OPT`, we are able to achieve significant performance benefits (up to 4x) on some benchmarks (both *adpcm* and *gsm* variants), performance within 130% of the CPU on 11 others, while 5 of the 26 benchmarks suffer slowdowns of 80% or more. Inspection of the procedures with the worst RF performance usually showed a loop-carried dependence involving a sequence of memory accesses with little other computation going on. With slightly better compiler analysis (or with a simple source code rewrite), the detrimental dependence cycle could be broken to get much better performance. The best-performing procedures, on the other hand, exploit Tartan's predicated, speculative execution. For example, *gsm*'s main loop has many small basic blocks: when merged into a single hyperblock, the loop has a large number of predicated operations that can be executed in parallel by the RF, gaining a significant performance advantage over the limited-issue processor. However, RF execution also suffers from bloated hyperblocks that combine too many paths—the critical path is determined by the worst case, and speculative memory operations consume bandwidth. We expect significant performance improvement when a smarter profile-based hyperblock formation algorithm is implemented.

In going from `OPT` to `NARROW`, most benchmarks suffer a very small performance degradation, since most of the computation happens on the RF. This indicates that it is perfectly reasonable to combine the RF with a narrow CPU to optimize for silicon resources and energy consumption. The two exceptions are *li* and *perl*. These benchmarks spend a significant portion of their execution time on the CPU. This is because both these benchmarks have calls to *longjmp*, and we currently execute all functions in the call

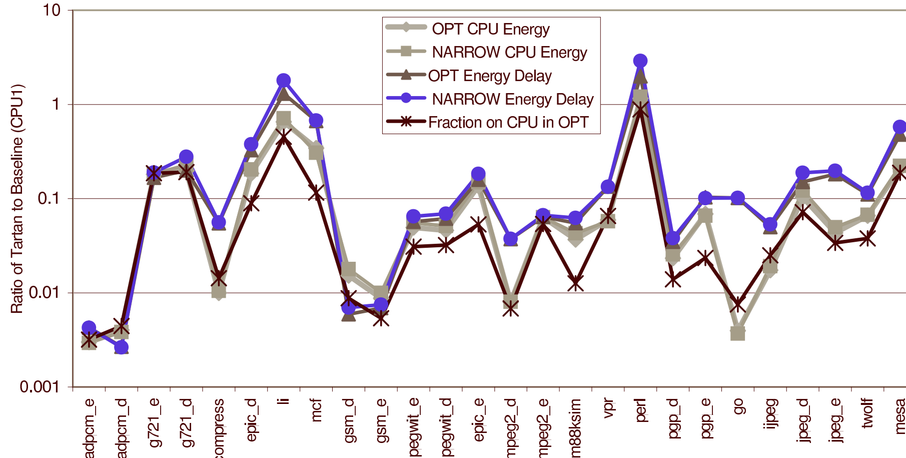

Figure 6. *Energy consumed on the CPU, and Energy-Delay, for the entire benchmark running on Tartan in the* `OPT` *and* `NARROW` *configurations. For comparison, the time spent on the Tartan CPU normalized to the* `CPU1` *execution time is also plotted. Note that the vertical axis is log-scale.*

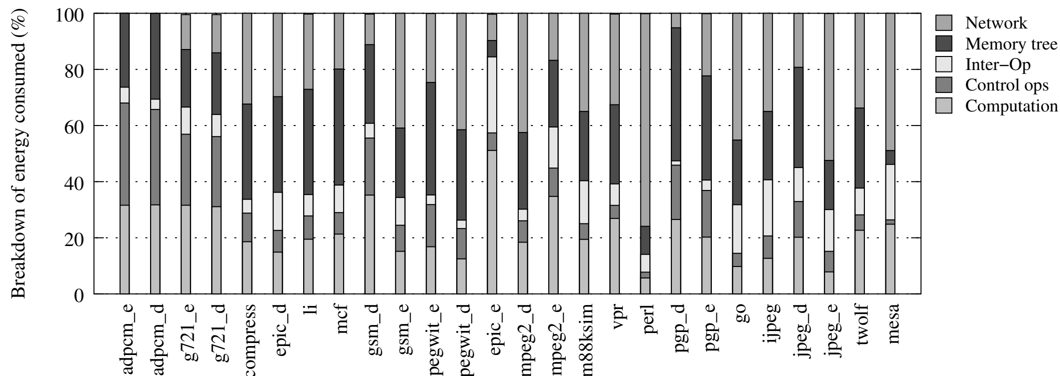

Figure 7. *Breakdown of energy consumption on the RF in* `OPT`

chain from the *setjmp* to the *longjmp* on the CPU. In moving from `OPT` to `NARROW`, the CPU-resident computation slows down by about 2x, proportionately hurting overall performance.

Implications: The results indicate that for a majority of benchmarks, Tartan's performance lags behind that of a superscalar from the same technology generation. This is not hard to understand: for control dependent codes found in unaltered C applications, performance will often be better on superscalar processors which combine speculation and dataflow (as shown in [4]); on SC fabrics, misspeculation across loop iterations or hyperblock boundaries is hard to effectively quash, restricting us to speculating only within a hyperblock. Better compiler analysis to reduce loop-carried dependences can improve performance and it is possible that many of these benefits can also be harnessed by simple source-code annotations. Many of the performance improvements in Tartan stem from executing predicated hyperblocks; these benefits can also be harnessed by a predicated microprocessor. Such a processor, however, will still suffer from having only a limited issue width. Another performance bottleneck is the latency of communicating with memory, which shows up in the memory access trees and network; this is explored further in Section 5.3. As expected, an unaggressive core does not impact performance significantly as long as most of the application is mapped to the RF.

5.2 Energy-delay

It has been previously demonstrated that a piece of code implemented as an asynchronous, spatially distributed dataflow machine has energy efficiency up to three orders of magnitude better than the same code executing on a CPU [8]. In this section, we evaluate how these benefits scale to a hybrid CPU–RF system like Tartan.

Figure 6 shows how much energy Tartan consumes on the core, and its Energy-Delay, normalized against the Energy and Energy-Delay of `CPU1` alone (a value of one means Tartan's core Energy or Energy Delay equals that of `CPU1` alone; smaller numbers are better). Tartan is run in the `OPT` and `NARROW` configurations. For comparison, the time spent on the CPU in Tartan's `OPT` configuration is also plotted, normalized to the total `CPU1` execution time. We observe that the Energy-Delay closely tracks the core energy, which in turn closely tracks the time spent on the CPU—since the RF is extremely energy-efficient, the overall energy efficiency is dominated by the energy efficiency of the core. When the core executes less (and hence consumes less energy), energy efficiency can be more than two orders of magnitude better than `CPU1`. For cases where a lot of computation occurs on the core (*li* and *perl*), Energy-Delay can be worse than `CPU1` (even though total energy consumed by Tartan is less than `CPU1`).

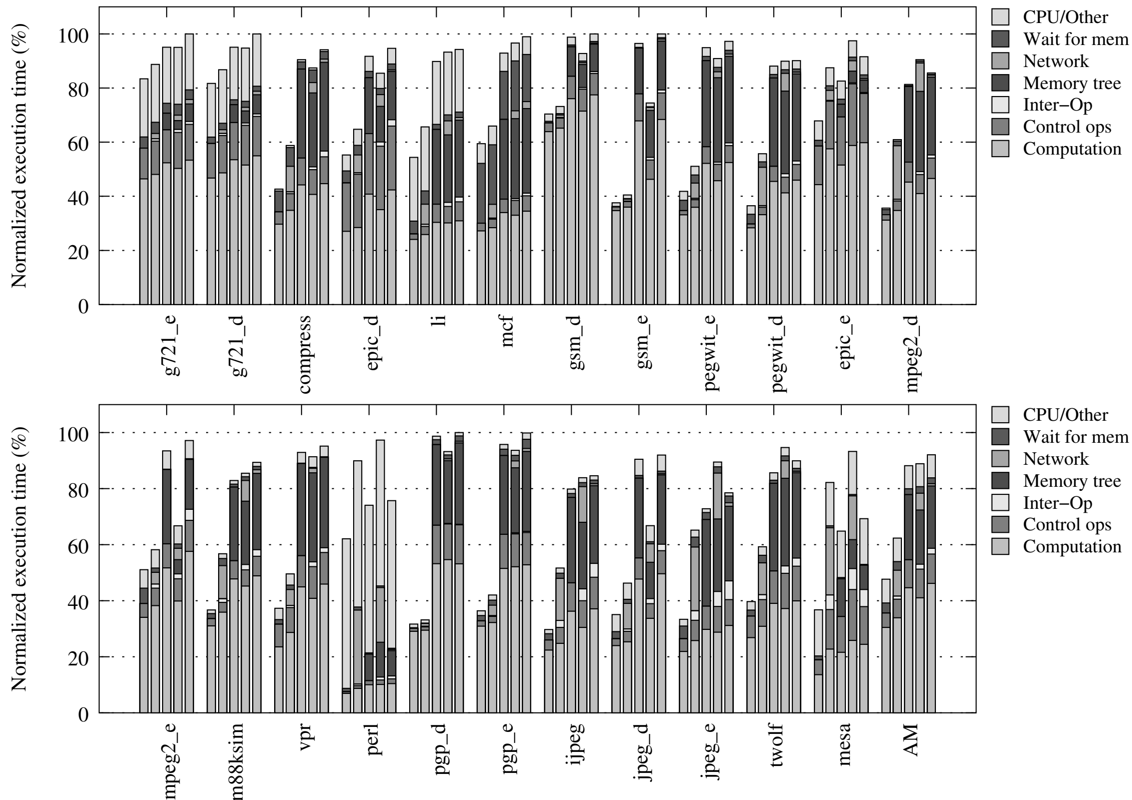

Figure 8. *Effect of communication costs on performance, normalized to* `OPT`. *The first bar is for 0 communication costs, the second is for 0 cost in the arbitration tree, the third for 0 cost in inter-node and NoC communication, the fourth for 0 congestion in the trees, and the fifth for a constant small delay of 0.5ns through the network. The* adpcm *benchmarks, which have very little communication, are not included.*

Figure 7 contains the breakdown of where energy is spent on the RF. Except for the smallest benchmarks, energy consumption is dominated by the communication hierarchy.

Implications: We see that the energy-delay benefits seen previously for kernels do translate over to whole programs, as long as the RF does most of the execution (note that all benchmarks had significantly better *power* and *energy* numbers compared to the CPU alone; however, the execution slowdown means that *energy-delay* is not always on par). Better partitioning of the application may let us achieve both performance and energy-delay benefits.

5.3 Effect of communication costs

Figure 5 shows that for most benchmarks, 40% to 80% of the RF's active time is spent only on communication, either between pages, for procedure calls, or for memory accesses. We identify opportunities for optimizing communication costs in this section. These results are summarized in Figure 8; all runs were with Tartan combined with CPU1 and normalized to the `OPT` configuration.

Recall that there are 3 types of communication involved: between operations in the same or different pages, in the dynamically arbitrated memory access trees, and over the global NoC. Typically, the first contributes very little to execution time; most of the latency is due to the memory trees and NoC. To measure the overall impact of communication, we first set all communication costs at all 3 levels to 0. As the first bar in Figure 8 shows, this reduces execution time by 24% to 74%, with an arithmetic mean of 54%.

We then set only the memory tree cost to 0, while retaining all other communication costs; conceptually, this corresponds to each memory operation having a dedicated channel to the corresponding NoC router. As the second bar shows, this accounted for most of the gains from 0 communication costs in all but 4 benchmarks (*epic_e*, *perl*, *jpeg_e* and *mesa*), with an average execution time reduction of 38%. In *epic_e*, inter-operation communication significantly reduces the exposed parallelism; in the latter three benchmarks, there are significant network transfer costs due to larger circuit size and the large number of memory accesses that traverse the NoC.

Next, we simulated a realistic memory tree but set all other communication costs to 0. This showed much less performance gain compared to the 0 tree-case (2 to 20% in most benchmarks with an average across all benchmarks of 12%) except in the aforementioned 4 benchmarks, where the gain was more significant.

To measure the impact of congested memory access trees due to many concurrently-active memory operations being placed close together, we set the congestion in the memory trees to 0, while keeping all other communication costs normal. Each memory operation still has to pay the cost of traversing the memory tree and the NoC to reach the LSQ, but does not have to suffer contention delays in the tree; such a situation would arise if potentially concurrent memory operations were placed in different memory trees. Compared to `OPT`, we see a reduction in delay of 16-36% in 6 of the 23 benchmarks; these benchmarks have a number of concurrent loads and stores that end up in the same tree and cause significant tree congestion. For 17 benchmarks, there was a difference of 20% or more between the 2nd and 4th bars, with an average across all benchmarks of 22%. This indicates that eliminating congestion in the trees is not enough—just the added delay of traversing (uncongested) trees reduces performance significantly.

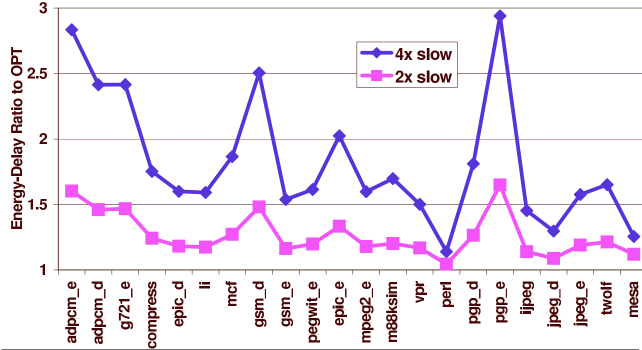

Figure 9. *Effect of slowing down compute and control nodes by 2x and 4x on Energy-Delay, normalized to* `OPT`*. Note that except for* perl *and* li*, the 4x line is still significantly better than* `CPU1`*.*

Finally, we simulated a case where there were realistic inter-operation costs, a realistic memory tree, but only a constant 0.5ns delay over the NoC. Physically, this corresponds to the case where an oracle predicts which procedure will be called next, and configures it close to memory and to its callers. The performance benefits over baseline in this case ranged from 0 for the smallest benchmarks (since they have very small network costs) to 35% for the largest circuit (*mesa*).

Implications : The above experiments highlight a key cost of spreading computation out in space: as things move farther away from memory, the cost of getting to memory and back becomes the dominant performance bottleneck. These costs show up most strikingly in the arbitrated access trees, but also through increased latency to traverse the NoC. This points to an important direction for future research: spreading memory out along with the computation to eliminate the long latency due to contention resolution and fabric traversals.

We also see that while dedicating a hardware unit to each logical operation saves us reconfiguration effort, it does mean that functions implemented on the RF will often be farther from memory than they would be if we could reconfigure the working set of pages close to the memory port.

5.4 Slower compute nodes

The timing and energy models for compute nodes used in our simulations are extracted from ASIC data. In the RF context, they will be worse due to the presence of muxes, which allow for programmability.[2] To measure the impact of slower computation, we ran two experiments, which slow down all compute and control nodes by a factor of 2 and 4 respectively. As discussed in Section 4, the slowdown of 4x represents a worst-case bound on the expected slowdown in a reconfigurable implementation, and the actual slowdown is expected to be closer to 2x or even less. For `OPT`, a 2x compute slowdown degrades performance by 2% to 48%, while for the 4x slowdown, performance degrades by 30% to 185%. The performance loss is much smaller than the slowdown factor because of overlap between compute and communication events. The largest slowdowns occur on the smaller benchmarks where there is very little communication to hide the greater compute latency; for the larger benchmarks, which are communication-bound, the performance degradation is comparatively small.

The impact on energy-delay is shown in Figure 9. The results highlight some important aspects of the RF design. The benchmarks impacted most are the ones that are computation-intensive (like *adpcm*, *gsm*), while there is a much smaller change

[2] The models for the communication structures, however, have been accurately estimated for the RF setting.

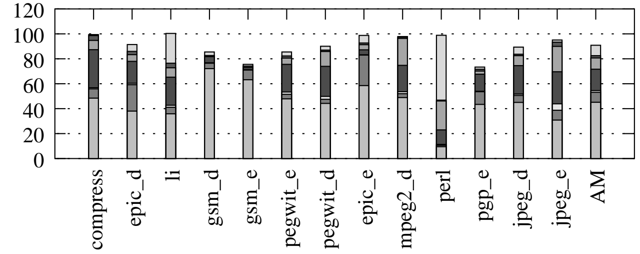

Figure 10. *Effect on performance of setting token communication cost to 0, normalized to* `OPT`*. The segmentation of bars is the same as Figure 8.* AM *is the Arithmetic Mean.*

in communication-intensive workloads. However, as shown in Figure 6, the computation-intensive benchmarks harness the energy benefits of the RF to the greatest degree, exhibiting two to three orders of magnitude improved energy-delay over `CPU1`. This large margin of benefit is very resilient in the face of slower computation nodes; in fact, even with the 4x compute slowdown, the energy-delay of Tartan is still significantly better than `CPU1` except for *perl* and *li*.

Implications: The energy and timing degradation in a typical FPGA compared to an ASIC is primarily due to high communication costs [32]. Shang et al. [33] show that 52% of the total power consumed in Virtex FPGAs is due to the interconnect, 22% is due to the clock distribution, and only 26% is due to computation logic. This is the prime motivation behind the hierarchical interconnect design in the RF. The lower levels of the hierarchy provide statically configured dedicated channels for higher performance, while higher levels of the hierarchy feature a shared interconnect for increased efficiency. The coarse-grained PEs and interconnect (compared to single-bit LUTs) amortize control signals over many data wires, reducing communication energy. Further, due to Tartan's asynchronous nature, the clock power is completely eliminated. Thus, an intelligent RF interconnect design is key to achieving scalable energy efficiency on large programs and future technology nodes.

5.5 Memory access and ordering costs

In Tartan, we achieve ordering between memory operations by passing *tokens* between them; operations determined to be independent by the compiler do not have token edges between them and may be active simultaneously. This contrasts with a scheme where the compiler provides each operation with its dependency information; this information is passed to an ordering unit which forwards the request to memory if all dependencies are satisfied [37]. To simulate such a scheme, we set the token communication costs in our simulator to 0; this is equivalent to having an oracle that could indicate to each memory operation when all its dependencies were satisfied so it could forward its request to the LSQ. This is more optimistic than [37], since we do not account for the storage, forwarding and runtime processing of dependency information. The results are shown in Figure 10; performance improves by an average of 8% when token costs are removed. The largest improvements occur in benchmarks where a large number of memory operations end up on the same memory access tree, increasing tree depth and congestion and slowing down token communication time. These benchmarks would benefit most from a more judicious tree construction that placed independent memory operations on separate trees.

Implication: Using tokens to manage the ordering of dependent operations incurs a small cost, but is not the primary bottleneck in performance. Distance to memory has a much bigger impact, and a more important source for performance improvement.

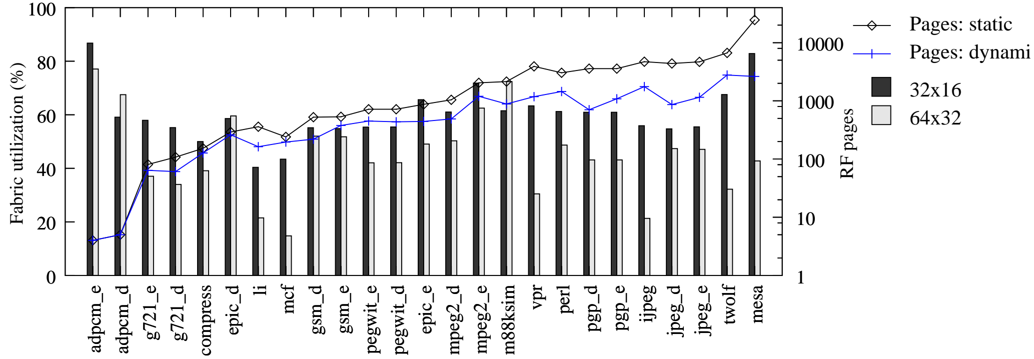

Figure 11. *Bargraph, left axis: RF utilization varying with page size. Linegraph, right axis (log scale): number of pages required to map the entire benchmark to Tartan's RF, and the number of pages that were actually touched during execution.*

5.6 Fabric utilization and page-size

All previous experiments used a page size of 32×16 bit-operations. Using a larger pagesize of 64×32 bitops did not affect performance significantly—in most cases lower delay through the NoC was offset by higher inter-operation communication cost over the intra-cluster interconnect. However, fabric utilization as measured by the number of bitops that were configured statically dropped significantly (see Figure 11). This is because our compiler generates many small hyperblocks which must each be given a whole page; we are improving our hyperblock selection algorithm so this can be avoided.

On Figure 11 we also see the number of pages (of size 32×16) each benchmark needs statically when mapped onto the RF with no resource sharing or virtualization, as well as the number of pages that were dynamically active in that particular run of the benchmark (the pages that were not dynamically active correspond to procedures that were not called, or hyperblocks within called procedures that were not executed). The largest benchmarks, *twolf* and *mesa*, statically require 6700 and 25000 pages respectively; all other benchmarks require less than 4700 pages. For *twolf* and *mesa*, the total number of pages active dynamically during the course of the execution is about 2700; this number is under 1500 for the remaining benchmarks. A page of size 32×16 bitops is estimated to occupy between 0.5mm^2 (based on scaled estimates from [39]) and 1.2mm^2 (based on [31]) in 90nm technology. With the smaller estimate, a 400mm^2 die can contain about 800 pages, which will be enough for the hardware working sets (measured at a granularity of pages) of 15 of the 25 benchmarks in Figure 11. Clearly, it is infeasible to map an entire large program onto the RF, but the much smaller working sets hold promise for virtualization. With technology shrinking, more pages can be placed on a single die; better CPU–RF partitioning can also lead to unused or rarely used functions not being mapped to the RF.

Implication: In the near future, most large programs cannot be completely implemented on the RF without virtualization. While virtualization implies reconfiguration costs, performance benefits may be available from configuring heavily communicating routines close to one another (see last experiment in Section 5.3), especially if the reconfiguration latency can be hidden. Another approach would be to use a more moderate form of SC which shares hardware resources between logical operations.

5.7 Defect tolerance

In future-generation fabrication technologies, defects are expected to be more commonplace than they are today, and every compute fabric could have multiple manufacturing defects. The asynchronous nature of the Tartan RF makes it easy to tolerate defects and parametric variations, since changes in the timing of individual nodes does not affect the global timing closure and hence the overall correctness and stability of the system. To measure the resilience of the RF to defects, we randomly introduced defects in 10% of the pages, rendering them unusable. This has the effect of stretching the circuit out by 10% and increasing communication costs. However, much of this increase is hidden by the parallelism available in the circuit and the impact on performance is very small, less than 2% for all our benchmarks.

Implications: The asynchronous, hierarchical and distributed RF tolerates defects without any significant impact on the speed of the compilation process or execution.

6. Related Work

Challenges in technology scaling are pushing future designs towards distributed, regular architectures. There is a wide range of choice in the design of such regular architectures—at one end of the spectrum are fine-grained reconfigurable architectures where each processing element is a LUT-like functional block; such designs have been used mostly in hybrid architectures where a standard CPU core has been augmented with a small amount of reconfigurable logic. At the other end, there are coarse-grained tiled architectures, where each processing element is a light-weight processor core. Our RF design fits roughly in the middle of this continuum: we have medium-grained reconfigurable resources that are organized as a tiled fabric.

Custom-computing, reconfigurable and hybrid systems. The most common examples of reconfigurable processors are FPGAs from vendors such as Xilinx and Altera. Some notable research projects are PRISM II [44], PRISC [28], NAPA [13], Chimaera [46], Garp [9], PipeRench [15], and many others.

As has been discussed in Section 2, the Tartan page architecture is based on PipeRench [15]. Like Tartan, some of the above architectures are hybrid, i.e., they combine a CPU core with reconfigurable logic. For example, PRISC [28] augments a RISC pipeline with a modest programmable function unit (PFU); its compiler takes assembly code compiled from standard C and looks for special patterns to transform automatically to exploit the PFU. Garp uses a synchronous reconfigurable array in a tightly-coupled coprocessor to accelerate inner loops; like Tartan, it has an automatic compilation path from standard C and uses spatial, aggressively speculative computation utilizing a SSA-based compiler IR extended to handle arbitrary memory accesses correctly [9]. The NAPA architecture [13] has a coprocessor that can accelerate either an inner loop or a region of straight-line code.

For their RF, however, none of the above approaches targets a true Spatial Computation model, with completely distributed computation and control: the reconfigurable logic is small, has a sin-

gle, global clock, and usually communicates over global buses. Another difference is that, in Tartan, the CPU and RF interact as peers, with the RF also able to invoke services from the CPU. This is unlike prior approaches where the RF acted as a slave to the CPU. A peer relationship enables more flexibility in hardware-software partitioning; for example, whole functions can be mapped onto the Tartan RF, while previous work has focused on smaller code segments and inner loops. This also has an impact on the CPU-RF interface design. While many previous efforts include the RF in the CPU datapath, we deploy the RF as a co-processor with ISA extensions for communication (as in some past projects, such as NAPA and Garp). This provides for greater flexibility and more opportunities to harness the benefits of the RF but also increases the latency of communication with the RF, imposing a lower-bound on the size of routines that can be profitably mapped to the RF. At the circuit-level, the Tartan RF is asynchronous, which naturally results in a dynamically-scheduled circuit. Finally, none of the hybrid approaches mentioned here (or the tiled architectures discussed next) were designed with power as a first-class concern; therefore, most architecture evaluations do not measure power consumption at all.

Tile-based Architectures. There has been a recent surge in research projects exploring distributed multi-core architectures involving a grid of processing elements connected by a mesh-like network. Some notable examples are Smart Memories [25], Imagine [29], RAW [38], TRIPS [30] and Wavescalar [37]. We will contrast the latter three with Tartan below.

A tile consists of an 8-stage inorder pipeline in RAW, a 5-stage pipeline in Wavescalar and a simplified single-issue pipeline in TRIPS; unlike Tartan, none of these architectures include reconfigurable functional units. Many instructions can share the same tile, and the mapping of instructions to tiles can occur dynamically and can change during the course of the execution, unlike the instance of Tartan studied in this paper where program operations are mapped statically to hardware units. Like Tartan, all three architectures include routed networks at the top level; however, the lower levels of their communication hierarchy consist of bypass networks and dynamically arbitrated buses in contrast to the statically configured FPGA-like interconnect we propose for Tartan.

The RAW compiler splits a program into multiple parallel threads, and statically schedules each thread on a tile: the compiler generates a schedule for instruction execution on each tile and for communication over two of the four global routed networks. The compilation and execution models for Wavescalar and TRIPS, on the other hand, are similar to Tartan: the compiler splits the program into multiple hyperblocks, and each hyperblock is independently scheduled across the fabric. Execution follows the dataflow order: operations are executed when their operands are ready. RAW achieves ordering of memory operations implicitly through the compiler-generated schedule; Wavescalar and TRIPS assign sequence numbers to memory operations in each hyperblock and enforce ordering at the LSQ. None of these architectures use the token-based mechanism used by Tartan.

Unlike Tartan, none of these architectures include a superscalar core to execute legacy and control-intensive codes. Also, unlike Tartan, they have not been designed to optimize for power or energy-delay, and instead focus primarily on performance.

The hierarchical composition of PEs in our RF resembles the HSRA fabric [40] to some degree, although that design used statically assigned interconnections at even the higher levels, which are less flexible and more inefficient for sparsely utilized channels. Our dynamic routing multiplexes the use of resources among many virtual channels.

Asynchronous FPGA Architectures. A number of asynchronous FPGA architectures have been proposed [39, 18, 16, 27, 45]. Unlike our medium-grained FPGA architecture, all of these previous architectures are fine-grained: the basic cell can compute functions of few inputs (typically less than 4). Like the Tartan RF, some previous architectures compute on bundled-data codes [16, 27], while the others either use dual-rail codes [45, 39], or can be configured to compute on both dual-rail and bundled-data [18]. In [27, 45, 39] each cell is pipelined, like in the Tartan RF; for the other architectures [18, 16] pipelining can be achieved using several cells, which is wasteful. None of these previous architectures are hierarchical or include complex memory support, and only a few provide for arbitration elements [27, 16].

7. Conclusions

We have described a general-purpose, hybrid system architecture, Tartan, that integrates a processor core with a reconfigurable fabric (RF). The RF's asynchronous design style is naturally power-efficient and resilient to process variations. We have evaluated this architecture on a number of single-threaded workloads and found that the spatial computing model of the RF provides excellent energy efficiency, is tolerant to manufacturing defects in future technology nodes and is able to harness fine-grained operation-level ILP.

The cost of laying computation out in space, however, requires a scalable interconnect design. We have shown how the Tartan RF tackles this problem by employing a hierarchical interconnect architecture, which allows the system to achieve high energy efficiency despite being constrained by a single, shared memory subsystem. These results will only improve on moving to a distributed memory/cache architecture. To harness the energy-efficiency benefits of the RF, however, the compiler must be able to aggressively partition the application so that most of the program execution occurs on the RF.

This paper explored the limits of an extreme form of spatial computation that allocates computation resources that are never shared, resulting in large area requirements. Although we expect the increasing transistor budget to alleviate this problem, in the near term we expect that virtualization of limited RF resources will be an important research direction. This path towards RF virtualization, with a starting point of complete spatial program layout on the RF, is the opposite of most previous efforts, which have incrementally moved more and more of the program's execution onto the RF. The goal in both approaches is to determine the optimal point in the spatial computation continuum, but the explorations start from different extremes and thus bring in different perspectives.

Acknowledgments

This work was supported by the NSF through ITR grant number CCR-0205523. The authors would like to thank David Koes, Mukesh Agrawal, David Andersen and the anonymous reviewers for many helpful comments and suggestions.

References

[1] M. Beck, R. Johnson, et al. From control flow to data flow. *Journal of Parallel and Distributed Computing*, 12:118–129, 1991.

[2] T. Bjerregaard and J. Sparsø. A scheduling discipline for latency and bandwidth guarantees in asynchronous Network-on-Chip. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*. IEEE, 2005.

[3] D. Brooks, V. Tiwari, et al. Wattch: a framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture (ISCA)*, pages 83–94. ACM Press, 2000.

[4] M. Budiu, P. V. Artigas, et al. Dataflow: A complement to superscalar. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 177–186, March 20-22 2005.

[5] M. Budiu and S. C. Goldstein. Compiling application-specific hardware. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 853–863, September 2002.

[6] M. Budiu and S. C. Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[7] M. Budiu, M. Mishra, et al. Peer-to-peer hardware-software interfaces for reconfigurable fabrics. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 57–66, April 2002.

[8] M. Budiu, G. Venkataramani, et al. Spatial computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14 – 26, October 2004.

[9] T. Callahan and J. Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. ACM, 2000.

[10] L. Carter, B. Simon, et al. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming, special issue*, 28(6), 2000.

[11] T. Chelcea and S. Nowick. Robust interfaces for mixed-timing systems. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 12-8, pages 857–873, 2004.

[12] R. Cytron, J. Ferrante, et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[13] M. B. Gokhale, J. M. Stone, et al. Co-synthesis to a hybrid RISC/FPGA architecture. *J. VLSI Signal Process. Syst.*, 24(2-3):165–180, 2000.

[14] S. C. Goldstein. The impact of the nanoscale on computing systems. In *IEEE/ACM International Conference on Computer-aided design (ICCAD)*, 2005.

[15] S. C. Goldstein, H. Schmit, et al. PipeRench: a coprocessor for streaming multimedia acceleration. In *International Symposium on Computer Architecture (ISCA)*, pages 28–39, May 1999.

[16] S. Hauck, S. M. Burns, et al. An FPGA for implementing asynchronous circuits. *IEEE Design & Test of Computers*, 11(3):60–69, 1994.

[17] J. R. Heath, P. J. Kuekes, et al. A defect-tolerant computer architecture: Opportunities for nanotechnology. *Science*, 280, 1998.

[18] N. Huot, H. Dubreuil, et al. FPGA architecture for multi-style asynchronous logic. In *Design, Automation and Test in Europe (DATE)*, pages 32–33. IEEE Computer Society, 2005.

[19] Intel Corp. *Intel Pentium M Datasheet*, January 2006.

[20] J. Kao, S. Narendra, et al. Subthreshold leakage modeling and reduction techniques. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 141–148, 2002.

[21] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'06)*, pages 21–30, February 2006.

[22] E. Larson, S. Chatterjee, et al. MASE: A novel architecture for detailed microarchitectural modeling. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, November 4–6 2001.

[23] C. Lee, M. Potkonjak, et al. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–335, 1997.

[24] S. A. Mahlke, D. C. Lin, et al. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Computer Architecture (ISCA)*, pages 45–54, Dec 1992.

[25] K. Mai, T. Paaske, et al. Smart memories: A modular reconfigurable architecture. In *International Symposium on Computer Architecture (ISCA)*, June 2000.

[26] B. J. Nelson. Remote procedure call. Technical Report CSL-81-9, Xerox Palo Alto Research Center, 1981.

[27] R. Payne. Self-timed FPGA systems. In W. Moore and W. Luk, editors, *International Conference on Field Programmable Logic and Applications (FPL)*, volume 975 of *Lecture Notes in Computer Science*, pages 21–35. Springer, 1995.

[28] R. Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 172–80. IEEE/ACM, November 1994.

[29] S. Rixner, W. J. Dally, et al. A bandwidth-efficient architecture for media processing. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 1998.

[30] K. Sankaralingam, R. Nagarajan, et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 422–433. ACM Press, 2003.

[31] H. Schmit, D. Whelihan, et al. Piperench: A virtualized programmable datapath in 0.18 micron technology. In *IEEE Custom Integrated Circuits Conference*, pages 63–66, 2002.

[32] L. Shang and N. Jha. High-level power modeling of CPLDs and FPGAs. In *International Conference on Computer Design (ICCD)*, pages 46–51, September 2001.

[33] L. Shang, A. S. Kaviani, et al. Dynamic power consumption in Virtex-II FPGA family. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 157–164. ACM Press, 2002.

[34] Standard Performance Evaluation Corp. *SPEC INT 95 Benchmark Suite*, 1995.

[35] Standard Performance Evaluation Corp. *SPEC INT 2000 Benchmark Suite*, 2000.

[36] I. Sutherland. Micropipelines: Turing award lecture. *Communications of the ACM*, 32 (6):720–738, June 1989.

[37] S. Swanson, K. Michelson, et al. Wavescalar. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 291–302, December 2003.

[38] M. B. Taylor, W. Lee, et al. Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams. In *International Symposium on Computer Architecture (ISCA)*, pages 2–13. IEEE Computer Society, 2004.

[39] J. Teifel and R. Manohar. An asynchronous dataflow FPGA architecture. *IEEE Trans. Computers*, 53(11):1376–1392, 2004.

[40] W. Tsu, K. Macy, et al. HSRA: high-speed, hierarchical synchronous reconfigurable array. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 125–134. ACM Press, 1999.

[41] G. Venkataramani, T. Bjerregaard, et al. SOMA: a tool for synthesizing and optimizing memory accesses in ASICs. In *International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 231–236. ACM Press, 2005.

[42] G. Venkataramani, M. Budiu, et al. C to asynchronous dataflow circuits: An end-to-end toolflow. In *International Workshop on Logic Synthesis*, June 2004.

[43] G. Venkataramani, T. Chelcea, et al. HLS support for unconstrained memory accesses. In *International Workshop on Logic Syntheis*, June 2005.

[44] M. Wazlowski, L. Agarwal, et al. PRISM-II compiler and architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16, Apr 1993.

[45] C. Wong, A. Martin, et al. An architecture for asynchronous FPGAs. In *Proceedings of Field Programmable Technology (FPT)*, pages 170–177, 2003.

[46] A. Z. Ye, A. Moshovos, et al. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable unit. In *International Symposium on Computer Architecture (ISCA)*, ACM Computer Architecture News. ACM Press, 2000.