2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

A New Dataflow Compiler IR for Accelerating Control-Intensive Code in Spatial Hardware

Ali Mustafa Zaidi
Computer Laboratory
University of Cambridge
Email: Ali-Mustafa.Zaidi@cl.cam.ac.uk

David Greaves
Computer Laboratory
University of Cambridge
Email: David.Greaves@cl.cam.ac.uk

***Abstract*—While custom (and reconfigurable) computing can provide orders-of-magnitude improvements in energy efficiency and performance for many numeric, data-parallel applications, performance on non-numeric, sequential code is often worse than what is achievable using conventional superscalar processors. This work attempts to address the problem of improving sequential performance in custom hardware by (a) switching from a statically scheduled to a dynamically scheduled (dataflow) execution model, and (b) developing a new compiler IR for high-level synthesis that enables aggressive exposition of ILP even in the presence of complex control flow. This new IR is directly implemented as a static dataflow graph in hardware by our prototype high-level synthesis tool-chain, and shows an average speedup of $1.13\times$ over equivalent hardware generated using LegUp, an existing HLS tool. In addition, our new IR allows us to further trade area and energy for performance, increasing the average speedup to $1.55\times$, through loop unrolling, with a peak speedup of $4.05\times$. Our custom hardware is able to approach the sequential cycle counts of an Intel Nehalem Core i7 superscalar processor, while consuming on average only $0.25\times$ the energy of an in-order Altera Nios IIf processor.**

Keywords: Dark Silicon, High-level Synthesis, Compilers, Custom Computing, Instruction Level Parallelism.

I. Introduction

Despite ongoing exponential growth of on-chip resources with Moore's Law, the performance scalability of future designs will be increasingly restricted. This is because the total *usable* on-chip resources will be growing at a much slower rate, due to the recently identified problem of Dark Silicon [1], [2]. To mitigate the effects of dark silicon, architects are increasingly employing custom (and reconfigurable) hardware in an effort to trade silicon area for efficiency and performance.

Custom hardware accelerators are already utilized as part of heterogeneous architectures in order to improve both efficiency and performance for data-parallel, numeric applications by multiple orders of magnitude [3]. Recently, researchers have also started looking into utilizing custom (and reconfigurable) hardware to improve the per-core energy efficiency for the general-purpose application domain [4], [5], [6], [7]. For this domain, *sequential* code frequently involves irregular memory accesses and complex control-flow, such as data-dependent branching, function calls & multilevel nested loops.

Unfortunately, while custom-hardware can efficiently accelerate numeric or data-parallel applications, general-purpose sequential code often exhibits much lower performance in custom hardware than a typical out-of-order superscalar processor [8], [6], [4]. Conservation Cores are able to almost match the performance of an in-order MIPS 24KE processor [4] while providing $10\times$ better energy-efficiency. Recent updates have improved this to being approx 20% faster than the MIPS baseline [5]. Similarly , the asynchronous static-dataflow hardware generated by Budiu et al. manages around $100\times$ better energy efficiency, but still proves to be consistently slower than a simulated 4-wide out-of-order superscalar processor [8]. On the other hand, DySER is consistently able to match or exceed the sequential performance of a 2-way out-of-order processor, but relies heavily on its host processor (to which it is tightly-coupled), to implement control-flow speculation, thereby severely restricting its efficiency advantage [7].

In this work, our goal is to enable much more pervasive utilization of custom or reconfigurable hardware for general-purpose computation in order to mitigate the effects of dark silicon. For this, we must (a) overcome the performance limitations on sequential code in custom hardware, (b) without compromising its inherent energy-efficiency, while (c) requiring minimal programmer effort.

The Importance of Sequential Performance: Esmaelzadeh et al. [1] identifed insufficient parallelism in applications as the primary source of dark silicon. Despite the decade-old push towards multicores, the degree of threaded parallelism in consumer workloads remains very low [9]. In this case, achievable speedup will be strictly constrained by the sequential fraction of an application due to Amdahl's Law. Given the complexity and effort of effectively parallelizing such non-numeric, general-purpose applications [10], [11], we assume that trying to expose any more explicit parallelism in such applications will be impractical using existing threaded programming models.

Even for server and datacenter applications that exhibit very high parallelism, per-thread sequential performance remains essential due to practical concerns not considered under Amdahl's Law – the programming, communication, synchronization & runtime scheduling overheads of many fine-grained threads can often negate the area & efficiency advantages of *wimpy* cores. Consequently, it is more cost effective to have fewer threads running on fewer *brawny* cores than to have a fine-grained manycore in most cases [12]. Add to this the fact that a vast amount of legacy code remains sequential, we find that achieving high sequential performance will remain critical for performance scaling for the forseeable future.

II. The Superscalar Performance Advantage

There are two main reasons that out-of-order superscalar processors are able to achieve higher performance on control-flow intensive sequential code [13]:

- Aggressive control-flow speculation to exploit ILP from across multiple basic-blocks, and
- Dynamic execution scheduling of instructions, approximating the dynamic dataflow execution model at runtime.

Control-flow speculation via Branch Prediction: Modern superscalar processors utilize aggressive branch prediction to relax the constraints imposed by control flow on ILP: branch predictors with very high accuracy ($\geq 95\%$) enable effective speculation across multiple branches, providing a much larger region of code from which multiple independent instructions may be discovered and executed out of order. To handle cases of branch mis-speculation, conventional processors make use of an in-order commit buffer (or re-order buffer). to selectively commit executed instructions in program order. When a misprediction is detected, executed instructions from the mispredicted paths can simply be discarded from this buffer, preserving correct program state.

Conversely, the key reason that custom-hardware implementations of sequential code have poor performance is the lack of an efficient & effective control-flow speculation mechanism. While it is possible to perform speculation in hardware on some *forward* branches through *if-conversion*, currently no mechanisms exist for safely speculating on *backwards* branches (i.e. loops), as it is difficult to implement mis-speculation roll-back and recovery mechanisms in spatial hardware without introducing a synchronization bottleneck.

High-level synthesis tools attempt to overcome this limitation by statically unrolling, flattening and pipelining loops in order to decrease the number of backwards branches that would be dynamically executed [14], but this can significantly increase the complexity of the centralized finite-state machines that implement the static schedule for the hardware datapath, resulting in very long combinational paths that can overwhelm any gains in IPC [15], [16]. Overcoming the performance limitations due to explicit control flow is the key issue that needs to be addressed for custom hardware to become performance-competitive with conventional processors on sequential code.

Approximation of Dynamic Dataflow Execution Model: In addition to aggressive control-flow speculation, superscalar processors employ dynamic execution scheduling, which helps in dealing with unpredictable behavior at runtime. Instructions are allowed to execute as soon as their input operands (as well as the appropriate execution resources) become available. For instance, in the event of a cache miss, only those instructions that are dependent on the stalled instruction would be delayed, while independent instructions continue to execute.

Processors can even have multiple instances of the same instruction (say from a tightly wound loop) in flight, with their results written to different locations via register renaming. Using renaming, contemporary processors are able to approximate the dynamic-dataflow model of execution, allowing them to easily adapt to runtime variability to improve performance.

On the other hand, custom hardware employs static execution scheduling, where the execution schedule for operations is determined at compile-time, and implemented at run-time by a centralized finite-state machine. This means that such hardware can only be conservatively scheduled for the multiple possible control-flow paths through the code, leaving it unable to adapt to runtime variability that may occur due to data-dependent control-flow, variable-latency operations, or unpredictable events such as cache misses.

A Case Study: The combination of these factors results in custom hardware exhibiting poor performance when implementing general-purpose sequential code via high-level synthesis. Consider for instance, the *internal_int_transpose* function from the *epic* Mediabench benchmark given in Figure 1. Budiu et al. identified this as a region of code that performs poorly when implemented as custom hardware [8].

At runtime, the inner do-while loop rarely executes more than once and never more than twice each time, while the outer-loop executes for a large number of iterations. The branch prediction logic in conventional processors adapts to this execution pattern and is effectively able to execute multiple copies of the outer loop, i.e. performing *outer-loop pipelining*, thereby effectively hiding much of the latency of the % and * operations in the inner loop.

```
/*==========================================
In-place (integer) matrix tranpose algorithm.
Handles non-square matrices, too!
==========================================*/
void internal_int_transpose(int* mat, int rows,
    int cols, int modulus ){
  int swap_pos, curr_pos, swap_val;

  for (curr_pos=1; curr_pos<modulus; curr_pos++)
      {
    swap_pos = curr_pos;
    do {
      swap_pos = (swap_pos * cols) % modulus;
    }
    while (swap_pos < curr_pos);

    if (curr_pos != swap_pos) {
      swap_val       =   mat[swap_pos];
      mat[swap_pos]  =   mat[curr_pos];
      mat[curr_pos]  =   swap_val;
    }
  }
}
```

Fig. 1. The *'internal_int_transpose'* function from the *'epic'* Mediabench benchmark

Conventional high-level snthesis tools would implement the control-data flow graph (CDFG) of this code, shown in Figure 2, as custom hardware. The blue operations belong to the inner do-while loop, the red operations belong to the subsequent if-block, while the yellow operations are the outer-loop iterator increment and condition-checking operations. Without branch prediction, each basic block (grey boxes) will be executed one at a time, in sequence. The exit predicates for the active block must be computed before control can *flow* to the next block.

One may attempt to alleviate the strict control-flow ordering somewhat by statically unrolling the loops. Unrolling

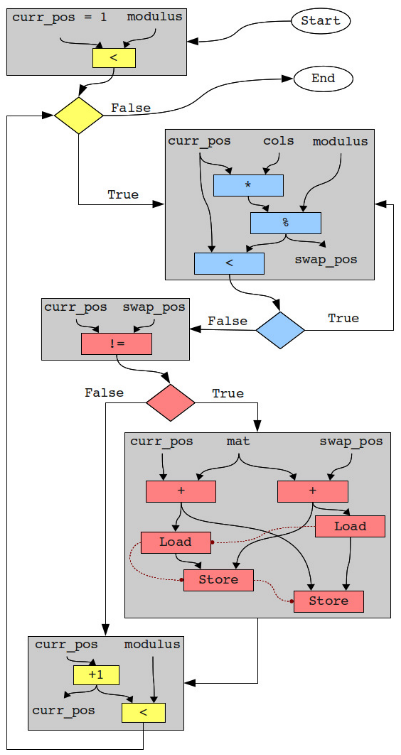


Fig. 2. The CDFG for *'internal_int_transpose'* function.

the innermost loop will not yield significant benefit as it rarely executes more than once. Unrolling the outer-loop would replicate the blocks that comprise it, including the inner-loop and the if-block. Due to the lack of mis-speculation recovery mechanisms for backwards branches and memory operations, neither the inner-loops nor the memory operations in the if-block can be executed speculatively. Thus, unrolling the outer-loop provides no benefit in this case, since the predicate computations of each basic block in the new sequence would still be on the critical path.

III. Improving Sequential Performance in Custom Hardware

To overcome the sequential performance issues of custom hardware, we propose two key changes during high-level synthesis:

- Instead of using a static scheduling based execution model for custom hardware, a dynamically scheduled execution model like Spatial Computation should be used [6], that implements code as a static dataflow graph in hardware.
- A new compiler IR is needed to replace the CDFG based IRs that are traditionally used for hardware synthesis. This new IR should be based on the Value State Dependence Graph (VSDG) [17], as it has no explicit representations of control-flow, instead only representing the necessary value and state dependences in the program.

Spatial Computation and dynamic execution scheduling: The Phoenix project undertaken at CMU proposed the *Spatial Computation* model, which combines the static-placement of existing custom hardware, with the dynamic execution scheduling of dataflow machines. Spatial computation is based on the static-dataflow execution model from the 1970s and 80s: execution of individual operations is dynamically scheduled, based on the status of its input and output edges. Spatial computation alone goes some way towards improving sequential code performance beyond statically scheduled hardware by not only exploiting greater parallelism through dataflow software pipelining, but by being more tolerant of variable or unpredictable latencies at runtime.

Overcoming control-flow constraints with the Value State Flow Graph: A new compiler IR for implementing spatial computation called the Value State Flow Graph (VSFG), has been developed based on the VSDG, but modified for direct implementation in hardware as a static dataflow machine. Unlike the CDFG, the VSFG represents control-flow only implicitly – there is no explicit routing of values based on predicates. Instead, the VSFG records only the *value* dependences between operations in a program, along with *state* dependences that serve to sequentialize side-effecting operations within a program and preserve correct ordering of memory operations during execution. Unlike the CDFG, there is no subdivision of operations into basic blocks, and consequently, no notion of flow of control from one block to another. The entire program is represented as an acyclic nested graph, so even loops are implemented without explicit cycles, by representing them as tail-recursive functions. Figure 3 presents the VSFG equivalent to the CDFG from Figure 2.

The VSFG is also a hierarchical graph – all loops and function calls are represented as nested subgraphs. From the perspective of their parent level in the graph hierarchy, nested subgraphs appear as ordinary dataflow operations with their defined dataflow inputs and outputs. The only difference being that nested subgraphs may exhibit a variable execution latency. Furthermore, just like regular dataflow operations, multiple such subgraphs may execute concurrently, so long as their dataflow dependences are satisfied. In Figure 3, the outer loop contains the inner-loop in a nested-subgraph represented by the block labeled *'Inner do-while Loop'*, the contents which are also shown in Figure 3 inside the shaded blue region on the right. Similarly, the next iteration of the outer-loop itself is represented as a tail-recursive call to the purple nested-subgraph marked *'Outer For Loop'*.

All of the operations in the graph retain their necessary dataflow or *value* edges from the CDFG (solid black arrows). Side-effect sensitive operations such as loads, stores, function call and loops subgraphs have two additional types of edges:

- An incoming and outgoing *state* edge (red dashed arrows) that pass a token between state-sensitive operations in order to enforce sequentialization of side effects in program order.
- An incoming *predicate* edge (green dotted arrows) that enforces correct control over the execution of operations, replacing the *flow* of control in the CDFG.

In the absence of an explicit *flow* of control between blocks, the execution of side-effects is controlled through the use of

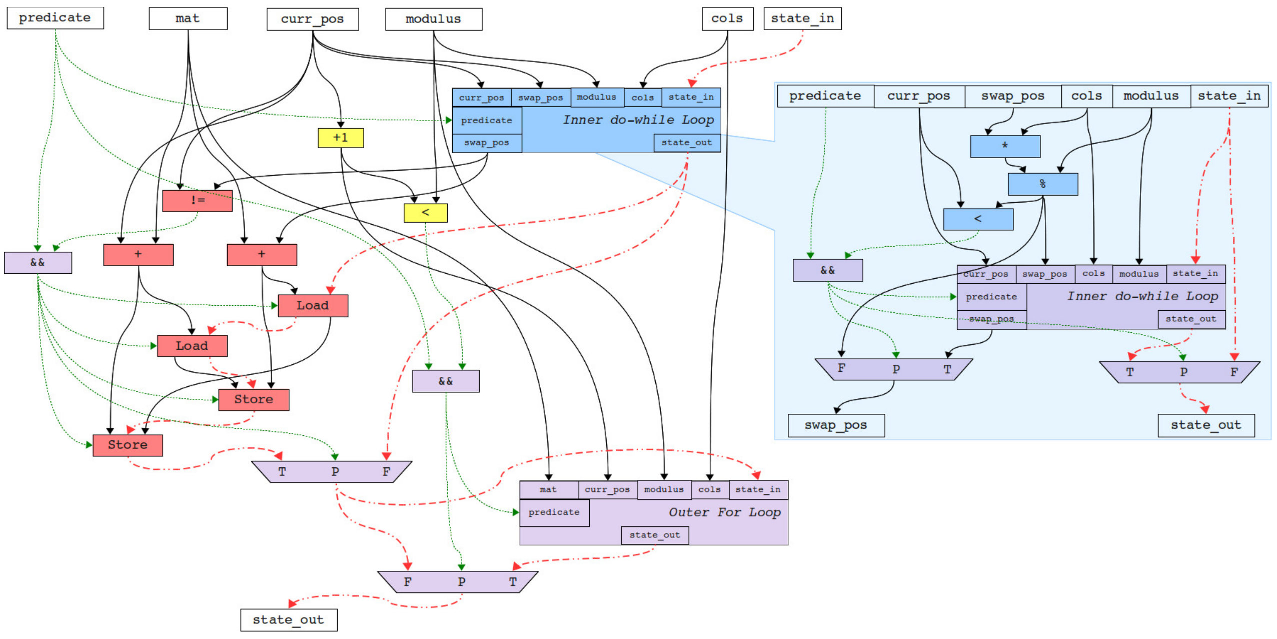


Fig. 3. The VSFG for the outer for-loop of *'internal_int_transpose'*, showing the inner do-while loop as a nested subgraph. The next iteration of both the outer and inner loops is also represented as nested subgraphs, essentially implementing loops as tail-recursive functions.

dataflow predicate inputs. Each basic block in the original CDFG will have an equivalent predicate expression implemented as additional operations in the VSFG. These predicate operations in conjunction with the multiplexers (both shown as purple blocks in Figure 3), serve to effectively convert all control-flow in the program into dataflow, while the state-edge does the same for side-effect ordering.

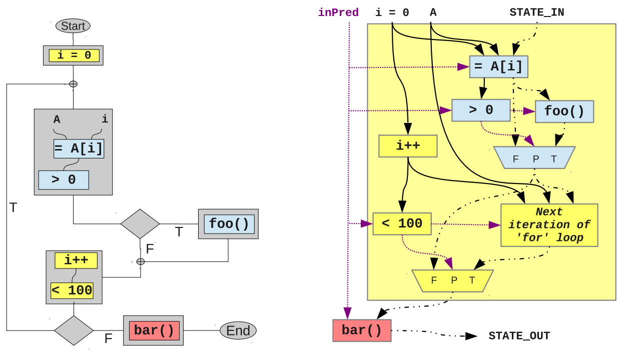
One key advantage of this lack of explicit control flow, combined with nesting of subgraphs is the ability to perform loop unrolling and pipelining at multiple levels of a loop nest. Any of the nested subgraphs in a VSFG can be *flattened* into the body of the parent graph. In the case of loops, flattening the subgraph representing the tail-recursive loop call is essentially equivalent to unrolling the loop. Figure 4 shows what happens when we flatten the *'Outer For Loop'* subgraph in Figure 3 once. Furthermore, as each loop is implemented within its own subgraph, this type of unrolling may be implemented within different subgraphs independently of others. Therefore, it is possible in the VSFG to exploit ILP within a loop nest by unrolling the inner loops independently of the outer loops. Note that in the actual hardware implementation, cycles must be reintroduced once the appropriate degree of unrolling has been done for each loop.

Thus for the example in Figure 1 that performs poorly when implemented as both custom hardware or even CDFG based spatial computation [8], by utilizing the VSFG as our dataflow IR, we are able to exploit *outer-loop pipelining* in the same manner as the superscalar processor, simply by flattening the nested-subgraph representing the outer-loop tail-recursive function call any number of times. The performance results for this code are listed as the *epic* benchmark in Figure 8.

Another advantage of having control flow converted in to boolean predicate expressions is the ability to perform

```
for (i = 0; i < 100; i++)
  if (A[i] > 0) foo();
bar();
```

Fig. 5. Example C Code for illustration purposes



(a) The CDFG for Figure 5. (b) The VSFG for Figure 5.

Fig. 6. The equivalent CDFG and VSFG for the code given in Figure 5.

control dependence analysis to identify regions of code that are control-flow equivalent and may therefore execute in parallel – provided all state and dataflow dependences are satisfied. Consider the *bar()* function in the code given in Figure 5 (the equivalent CDFG is given in Figure 6a). Despite aggressive branch prediction, a superscalar processor will not be able to start executing *bar()* until it has exited the loop. Similarly, when the *if* branch is predicted-taken, the superscalar processor must switch from executing multiple dynamically unrolled copies of the for-loop and instead focus on executing the control-flow within *foo()*. This is because a conventional processor can only execute along a single flow of control [18].

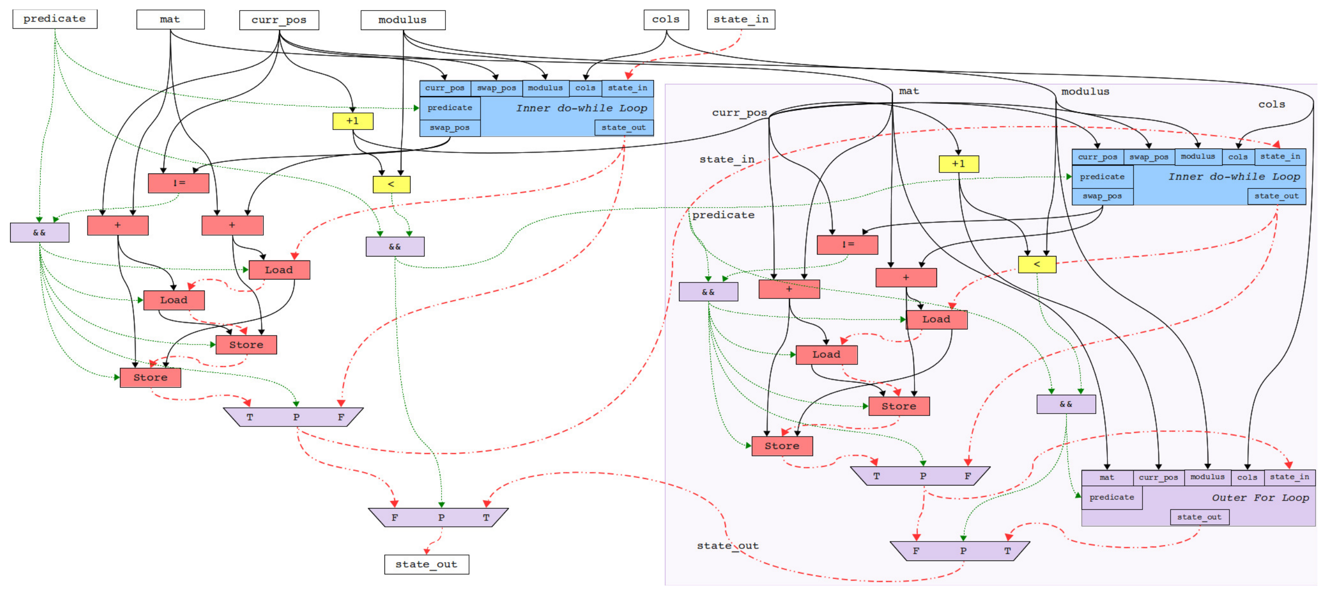


Fig. 4. Loop *unrolling* is implemented by *flattening* the loop's tail recursive call subgraph a number of times. Here, the VSFG of the *'internal_int_transpose'* outer for-loop, is unrolled once, implementing two copies of the inner loop. Each loop in a loop nest may similarly be flattened/unrolled independently of the others to expose loop parallelism.

The VSFG on the other hand can use control dependence analysis to identify the control-flow equivalence between the contents of the for-loop and the *bar()* function, and so long as the dataflow and state-ordering dependences are resolved, *bar()* may start executing in parallel with the for-loop. Similarly, the contents of the *foo()* subgraph can also execute in parallel with its parent graph. If we combine this concurrency with loop unrolling as shown in Figure 7, it becomes possible in to execute multiple copies of the loop and *foo()*, in parallel with the execution of *bar()*! This ability to execute multiple regions of code concurrently is equivalent to enabling execution along multiple flows of control, and can potentially expose greater ILP than even a superscalar processor [18], [19].

IV. Evaluation Methodology

The objective of the following experimental evaluation is two-fold: (1) evaluate the potential of the VSFG for statically exposing ILP from control-intensive sequential code, relative to equivalent, conventional CDFG-based custom hardware, and (2) understand the energy and area cost incurred for the observed improvements in ILP.

To evaluate our IR and execution model, we implemented a HLS toolchain to compile LLVM IR to custom hardware. We present results for 3 versions of our hardware: VSFG_0 has no loop unrolling/flattening, VSFG_1 has all loops unrolled once, and VSFG_3 has all loops unrolled thrice. At present, our tool-chain is an early-stage prototype, and as such can only compile applications with some constraints – currently there is no support for C language *structs* or multi-dimensional arrays. Additionally, like most HLS tools, support for general recursion and dynamic memory-allocation is also restricted. Although this has limited our choice of benchmarks applications, we have selected six benchmarks from the CHStone benchmark suite [20], and two home grown: *bimpa*, a spiking neural-network simulator, and *epic*, the micro-benchmark identified as being difficult to accelerate in spatial hardware by Budiu et al. [8], and discussed in Section II. Four of these 8 benchmarks exhibit sufficiently complex behaviour to justify this preliminary analysis: *epic*, *bimpa*, *adpcm* and *dfsin*.

To provide a baseline for comparison, we use the LegUp HLS tool [21] to generate CDFG-based, statically-scheduled custom hardware. The input LLVM IR to both toolchains was compiled with -O2 flags, and no link-time inlining or optimization. Both tools were run with operation-chaining disabled – meaning that generated hardware is fully pipelined, with each instruction in the IR having its own output register instead of being merged as combinatorial logic with a predecessor or successor. This was done because enabling operation-chaining would have masked the ILP improvements by reducing the degree of pipelining in the generated hardware to match the achievable operating frequency.

We also compare cycle counts against two conventional processors: a Intel Nehalem Core i7, as well as an Altera Nios II/f. The former was simulated using the Sniper interval simulator [22], while the latter was implemented, along with all of our generated hardware, on an Altera Stratix IV FPGA. The Core i7 used perfect L1 caches (100% hit rate), and a hit latency of 1 cycle, while the Nios was configured to access local RAM blocks on the FPGA, again with 1 cycle access latency. This is done to match the 1 cycle BRAM access latency of the generated custom hardware.

We focus on utilizing 'cycle-counts' as a measure of performance because they provide a good analogy for the degree of ILP exploited across different architectures. It would otherwise be difficult to provide a meaningful comparison of circuits implemented as ASIC hardware, or on an FPGA, with a full-custom superscalar processor (with its $3-30\times$ frequency advantage). It also allowed us to evaluate the potential of our IR without being too confounded by low-level implementation

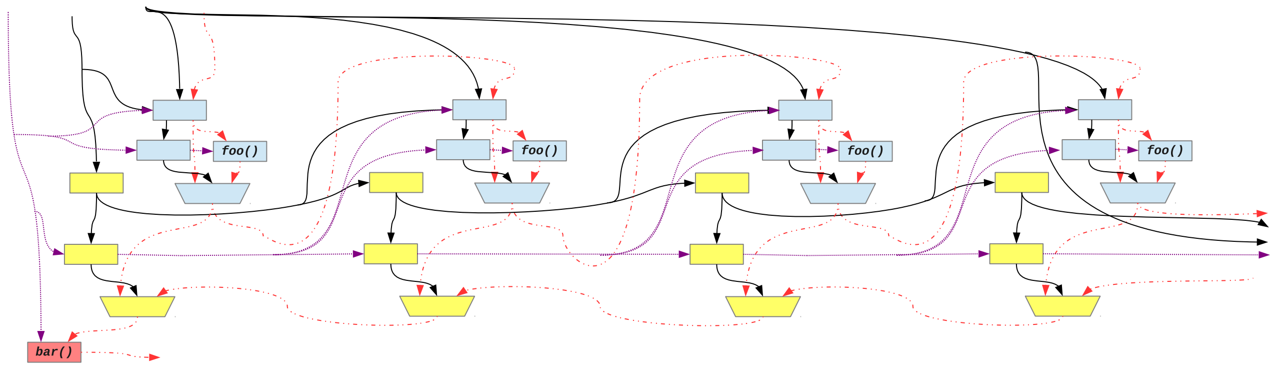


Fig. 7. The VSFG from Figure 6b with the loop unrolled 4 times.

details and design trade-offs (such as optimising designs for high-frequency) at this stage of our work.

As it has proven difficult for us to provide a fair comparison of power and energy efficiency between our generated hardware (implemented on an FPGA) and full-custom processor like the Intel Core i7, we present an energy cost comparison with the Nios soft-processor only, with the assumption that the in-order six-stage pipeline of the Nios should provide much higher energy efficiency than the out-of-order Core i7, if both were implemented using the same full-custom VLSI process. One may refer to the empirical relationship between power and performance for sequential processors presented by Grochowski et al. [23]: $Power = Perf^{\alpha}$, where $\alpha = 1.75$, and use our cycle-count results to estimate the difference in both power and efficiency between the Nios and the Core i7.

V. Results & Discussion

Performance (Cycle Counts): Figure 8 shows the normalized cycle count results for the various benchmarks for LegUp and the three VSFG versions. In most cases, the baseline VSFG_0 configuration is able to achieve a lower cycle count than the hardware generated by LegUp. Further performance improvements beyond VSFG_0 are achieved through aggressive multi-level loop unrolling. Figure 8 shows that VSFG_3 achieves 35% lower cycle counts than LegUp, by trading area for performance.

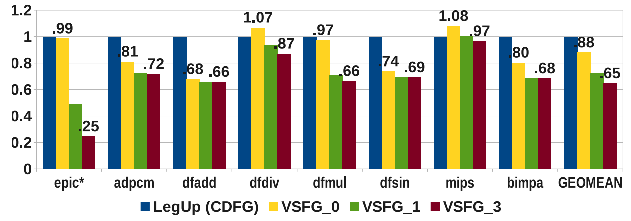
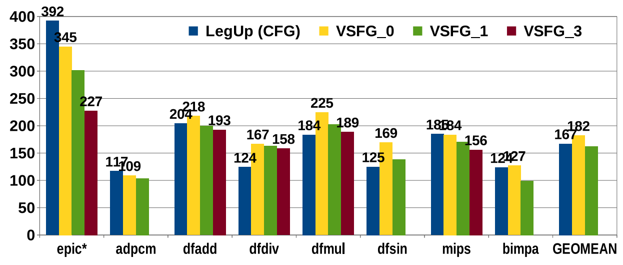
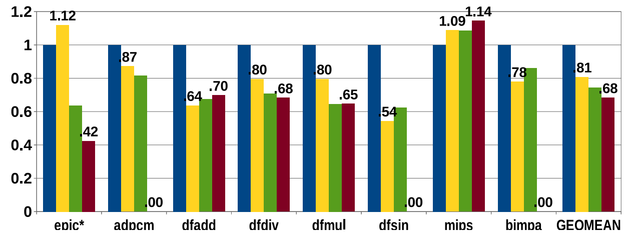


Fig. 8. Performance Comparison (Cycle Count) Normalized to LegUp.

Figure 9 compares LegUp & VSFG cycle counts to those of the Core i7 and Nios II/f processors for the 4 complex benchmarks (*epic*, *adpcm*, *dfsin*, and *bimpa*) benchmarks, showing the performance advantage of the superscalar Core i7 over the in-order Nios, as well as the CDFG-based LegUp. With unrolling, VSFG_3 is able to approach or exceed Core i7 cycle counts for all benchmarks with the exception of *bimpa*. Furthermore, for all benchmarks except *epic*, performance gains stagnate quickly between VSFG_1 and VSFG_3. The reason for this stagnation, and the limited performance on the memory-intensive *bimpa*, is the lack of memory parallelism or reordering in the current VSFG: all memory operations are constrained by the state-edge to occur strictly in sequential program order, whereas the Core i7 is able to dynamically disambiguate, re-order and issue multiple memory ops each cycle. For non-trivial code, exposing memory-level parallelism through memory disambiguation is essential for improving performance in modern superscalar processors. Although we have focused only on overcoming control-flow in this work, effective memory disambiguation must also be incorporated in order to truly match superscalar performance for general-purpose applications. In subsequent work, we plan on incorporating static memory disambiguation (alias-analysis) into our toolchain to try and address this limitation without incurring the energy cost of dynamic disambiguation.

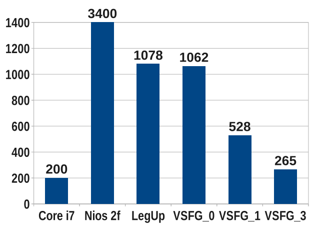


(a) f_{MAX} for benchmarks. Altera Nios II/f runs at 290 MHz.

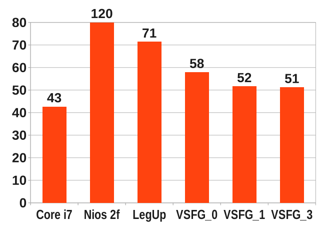


(b) Normalized Delay (f_{MAX} × *cycle count*) for benchmarks.

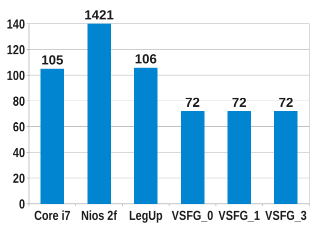
Fig. 10. Frequency (f_{MAX}) and Delay comparison of LegUp and VSFG custom hardware. note that some values for VSFG_3 are missing, as these circuits were too large to fit in our target FPGA. These have been removed from the Geomean calculation for VSFG_3.



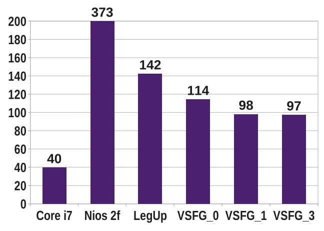
(a) epic (×1K cycles)



(b) adpcm (×1K cycles)



(c) dfsin (×1K cycles)



(d) bimpa (×1M cycles)

Fig. 9. Performance Comparison (Cycle Count) vs an out-of-order Intel Nehalem Core i7 processor, and an Alteral Nios IIf in-order processor.

Clock Frequency, Delay & Area: The Nios II/f achieves an f_{MAX} of 290MHz. Unfortunately, despite the high degree of pipelining by both LegUp and in the VSFG, clock rates are inversely related to the size of the circuit being implemented. For both tools, all of the memory operations distributed across each generated circuit are connected to a single, centralized memory controller. The critical-path wire-length thus increases with the total number of memory operations in the circuit, and are thus often below the achievable f_{MAX} for the carefully optimized Nios II/f design. Nevertheless, the baseline VSFG_0 configuration achieves 15MHz higher frequencies on average than LegUp. This was expected, as the dynamically scheduled VSFG is fully decentralized, and does not have a centralized FSM, unlike LegUp.

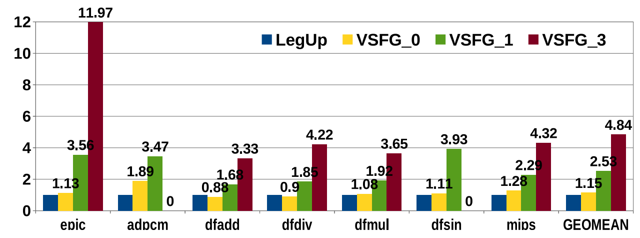
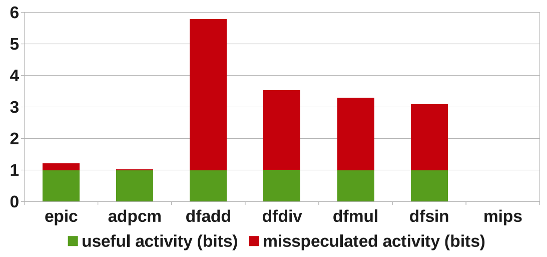


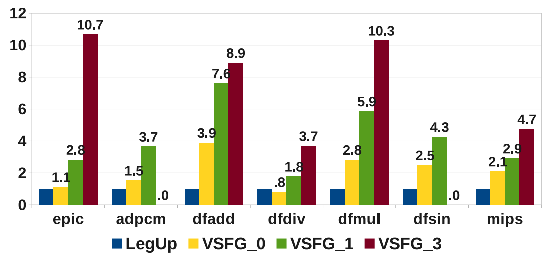
Fig. 11. Resource requirements comparison (number of FPGA LUTs) vs LegUp.

Resource requirements for both LegUp & VSFG_0 are very similar, as shown in Figure 11, with the VSFG_0 exhibiting an average 15% resource overhead over LegUp. We expect that this may be reduced with further optimization of our toolchain. However, requirements for VSFG_1 and VSFG_3 grow dramatically over VSFG_0, particularly for benchmarks with nested loops. This is because each loop in a nest is unrolled independently. For instance, for *epic* (Figure 4), VSFG_3 will have 4 copies of the outer for-loop. As the inner-loop is also unrolled 4 times, there will be a total of 16 copies of the inner loop in the circuit, leading to a 12× area overhead over LegUp! We unroll all loops blindly here in order to evaluate the limits of achievable instructions per cycle (IPC). But in practice, it would be essential to carefully balance IPC improvement with the area and frequency cost incurred from unrolling each loop in the code. For now, we leave considerations of area & frequency optimization for future work, as our focus is on evaluating the ILP exposition potential of our new IR.

Power Dissipation: We instrumented the generated VSFG hardware to estimate the amount of circuit activity overhead due to speculative execution. Figure 12a presents the



(a) Activity overheads due to aggressive speculation in the VSFG.



(b) Power dissipation comparison of VSFG, normalized to LegUp.

Fig. 12. Estimated Power Dissipation Comparison vs LegUp. The overheads for the VSFG are proportional to the increase in activity due to aggressive speculation.

misspeculated activity overheads for 6 of the benchmarks (measuring the average number of bits switching per-cycle), while Figure 12b presents the estimated power dissipation for each circuit configuration. Power is measured using Altera's PowerPlay Power Analysis tool, using activity ratios generated for each circuit from ModelSim simulation. For simplicity we assume a 250 MHz clock rate for all circuits, including the Nios II/f. Unfortunately, misspeculated activity results for *mips* were unavailable due to problems with our prototype compiler toolchain, and power results for VSFG_3 versions of *adpcm* and *dfsin*, and all versions of *bimpa* were unavailable as these circuits were too large to fit in our target FPGAs.

VSFG_0 mostly exhibits higher power dissipation than LegUp, echoing the pattern for speculation overhead in Figure 12a – *dfadd* for instance has both the highest speculation overhead and the highest power dissipation for VSFG_0, while both *epic* and *adpcm* have much lower of both. It may be possible to mitigate the energy overheads of speculation by making use of the hierarchical nature of the VSFG. Mutually exclusive control-regions in the VSFG may be extracted into their own sub-graphs. Speculative execution of these subgraphs would be enabled only after code-profiling determines that these regions execute frequently, and therefore may affect the performance critical path. This type of analysis and optimization would be based on existing work on profiling driven hyperblock formation [24]. We plan on implementing careful subgraph extraction and/or refactoring into our toolchain to explore how far we can mitigate the energy overheads of speculation without compromising performance.

The additional increase in power dissipation for VSFG_1 and 3 is driven primarily by non-computational overheads: dynamic power increases because clocking overhead grows proportionally to the registers in the circuit, while static power increases in proportion to the increased circuit size. For a more realistic implementation, loop unrolling ought to be applied only when performance can be improved, and the degree of loop unrolling should be carefully balanced against the power and area overheads that may be incurred.

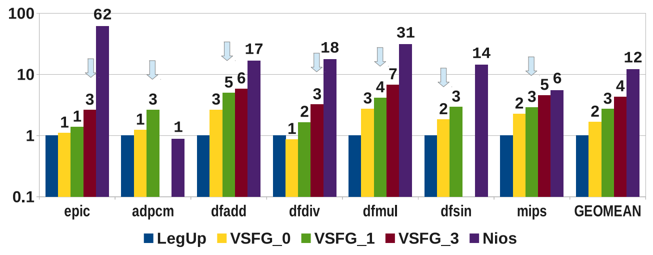


Fig. 13. Energy Consumption comparison vs LegUp and an Altera Nios IIf in-order Processor.

Energy Cost: Combining our cycle-count and power measurements, Figure 13 presents an energy cost comparison between LegUp, each VSFG version, as well as pure software implemented on the Altera Nios. Again, we assume that all hardware is able to run at 250MHz – this is not an unreasonable assumption, given recent work on pipelining the memory infrastructure in recent literature ([5], [25], [6]). The mean energy cost for the 35% higher performance of the VSFG_1 and VSFG_3 configurations is between $3 - 4\times$ over LegUp. However, it is also about $0.25\times$ the energy cost of the in-order Nios II/f processor. Therefore, despite our implementing no optimizations towards minimizing energy, the VSFG is able to approach superscalar performance levels for our set of benchmarks, while incurring only a quarter of the energy cost of an in-order processor.

The objective of this analysis was to explore the achievable performance of our new IR, without much effort given to optimize energy or area. We expect further improvements in both performance and efficiency to be possible once we incorporate alias-analysis to increase memory-level parallelism, and implement profile-driven sub-graph speculation and loop-unrolling.

VI. Related Work

Our work is related to the general class of Explicit Data-Graph Execution Architectures [26], of which there are several notable examples [27], [28], [29], [6]. The closest relative to our work is the Pegasus IR by Budiu et al. [30], [6], the original attempt at full program compilation to static dataflow custom hardware. Programs are compiled to CDFGs, optimized for performance using aggressive loop unrolling and hyperblock formation [24] to increase ILP from forward branches. Alias analysis is employed to expose memory level parallelism in the IR. Budiu et al. identified two primary bottlenecks in their work: significant latency overheads due to a deeply pipelined memory arbitration tree that is necessary to support parallel memory requests [6], as well as performance constraints due to complex control flow [8]. Other projects have attempted to optimize the memory access network for custom hardware by either optimizing for the most frequent accesses [31], partitioning and distributing memory [25], or incorporating cache-like structures [32], [5]. Tartan, a reconfigurable architecture for spatial computation was also developed [33].

In our work, we focus instead on overcoming the control flow hazards, particularly those identified in [8]. We've assumed an unpipelined memory bus with a single cycle access latency, but this comes at the cost of limiting achievable clock frequencies, particularly for larger programs. We aim to incorporate features of the aforementioned prior work into our memory architecture in the future to improve our clock rates and exploit locality for further efficiency gains.

The Wavescalar project developed a scalable, high ILP dynamic dataflow architecture [27]. Applications were compiled from high level languages to the dataflow Wavescalar ISA for execution. Performance was comparable to an Alpha EV7 processor, with 20% better performance per unit area. As the primary goals were performance and scalability, the energy costs of their implementation were not presented. Wavescalar did not implement control-flow speculation, relying on predication only for control-flow. Like us, it too relied on exploiting multiple-flows of control to improve performance.

More recent work addressing the Dark Silicon problem through the use of HLS to generate custom hardware is being undertaken by the GreenDroid project [4], [5], [34]. Identified *hot* regions of code are compiled to custom hardware *c-cores*, emphasizing efficiency over performance. As with other tools, code CDFG is compiled to statically scheduled custom hardware and interfaced as a coprocessor with an in-order MIPS 24KE host processor. Average c-core performance is equivalent to the in-order host, while providing $10\times$ better energy efficiency for each such code region. Performance has been improved in successive iterations of their work through the exploitation of 'selective depipelining' to reduce register delays and 'cachelets' to improve memory access times [5]. Until we can develop an EDGE architecture of our own like Tartan or Wavescalar, this is the architecture model we envision custom cores generated by our tool will be used in.

Previous notable work in improving the performance of control-intensive code in custom hardware was undertaken by Gupta et al under the SPARK compiler project [35]. The SPARK compiler utilized a CDFG-like IR called the Hierarchical Task Graph (HTG), and implemented speculative code-motion techniques permitting earlier execution of instructions my moving them into earlier basic-blocks, within an acyclic CFG region. Both the LegUp and VSFG compilers rely on the LLVM infrastructure, that possesses many built-in optimization and transformation passes that perform some similar (though not the speculative) code motion optimizations before ultimiately compiling to custom hardware. Furthermore, our VSFG-based hardware relies on dynamic execution scheduling, combined with predicate promotion to perform aggressive speculative execution of instructions, obviating the need for such transformations. It would be interesting in the future to incorporate such speculative code-motion transforms into LegUp and see how they might improve its performance on control-intensive code.

Aside from work by Budiu et al, PACT's XPP (eXtreme Processing Platform) project developed a CGRA that also utilized a static-dataflow execution model [36]. However this architecture and associated XPP-VC compiler only supported a subset of the C-language, and focused on compiling numeric/multimedia kernels [37]. Each of these static-dataflow approaches, including ours, could stand to benefit from improved performance through the incorporation of work on *cancel-tokens* by Gädke and Koch [38]. Cancel-tokens allow for termination of execution along misspeculated paths of a forward branch once the branch predicate is determined, and can accelerate execution when such branches lie within a loop.

Several other projects have also attempted to break the ILP Wall by exploiting multiple flows of control. Notable examples include the Multiscalar architecture [39], as well as a large body of work on Speculative Multithreading [40].

VII. Conclusion and Future Work

Conclusion: In order to address the constraints on sequential code performance in custom hardware, we combine a static-dataflow execution model with a new compiler IR based on the Value State Dependence Graph (VSDG) [17]. The hierarchical and control-flow agnostic nature of the VSDG not only enables the exploitation of ILP from across multiple levels of a loop nest, but also enables control-dependence analysis and execution of multiple flows of control, giving the potential for exploiting more ILP than even complex superscalar processors.

Our results show a performance improvement for our IR of as much as 35% over LegUp, at a $3\times$ higher average energy cost. Our cycle-counts approach a simulated Intel Nehalem Core i7, while incurring only $0.25\times$ the energy cost of an in-order Altera Nios IIf processor. These improvements are achieved without any significant optimizations to the IR and hardware that could further improve both efficiency and performance. Although much additional work is needed in order to implement a more robust compiler, capable of executing a larger set of benchmarks, as well as to optimize the generated hardware for area, power and frequency, we believe that the new IR, (coupled with the spatial computation execution model) is a promising step towards improving sequential code performance for custom hardware. It is hoped that our work will eventually contribute to more pervasive utilization of custom hardware and (coarse-grained) reconfigurable architectures, and thereby aid in mitigating the effects of dark silicon.

Future Work: The lack of memory level parallelism is the most significant performance bottleneck in our current implementation. Processors like the Core i7 aggressively reorder and parallelize memory accesses, having many tens of such instructions in flight. Consequently, the Core i7 still has a performance advantage for memory intensive benchmarks. To address this limitation, our next step will be to use alias analysis to parallelize memory operations [6], as well as exploit novel memory architectures in hardware [5], [25] to improve locality, concurrency and energy efficiency in our memory infrastructure.

References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108

[2] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, "The greendroid mobile application processor: An architecture for silicon's dark future," *Micro, IEEE*, pp. 86–95, March 2011.

[3] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 37–47. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815968

[4] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. Taylor, "Conservation cores: reducing the energy of mature computations," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 205–218, 2010.

[5] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. B. Taylor, "Efficient Complex Operators for Irregular Codes," in *HPCA 2011: High Performance Computing Architecture*, 2011.

[6] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XI. New York, NY, USA: ACM, 2004, pp. 14–26. [Online]. Available: http://doi.acm.org/10.1145/1024393.1024396

[7] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Proceedings of 17th International Conference on High Performance Computer Architecture (HPCA)*, 2011.

[8] M. Budiu, P. Artigas, and S. Goldstein, "Dataflow: A complement to superscalar," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, march 2005, pp. 177 –186.

[9] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 302–313. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816000

[10] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006. [Online]. Available: http://dx.doi.org/10.1109/MC.2006.180

[11] W.-m. Hwu, S. Ryoo, S.-Z. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Implicitly parallel programming models for thousand-core microprocessors," in *Proceedings of the 44th annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 754–759. [Online]. Available: http://doi.acm.org/10.1145/1278480.1278669

[12] X. Liang, M. Nguyen, and H. Che, "Wimpy or brawny cores: A throughput perspective," *Journal of Parallel and Distributed Computing*, no. 0, pp. –, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731513001160

[13] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: is it speculation or dynamism?" in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451143

[14] P. Coussy and A. Morawiec, *High-Level Synthesis: from Algorithm to Digital Circuit*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[15] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Loop shifting and compaction for the high-level synthesis of designs with complex control flow," in *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, ser. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 10 114–. [Online]. Available: http://portal.acm.org/citation.cfm?id=968878.969079

[16] S. Kurra, N. K. Singh, and P. R. Panda, "The impact of loop unrolling on controller delay in high level synthesis," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '07. San Jose, CA, USA: EDA Consortium, 2007, pp. 391–396. [Online]. Available: http://dl.acm.org/citation.cfm?id=1266366.1266449

[17] N. Johnson and A. Mycroft, "Combined code motion and register allocation using the value state dependence graph," in *Proceedings of the 12th international conference on Compiler construction*, ser. CC'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 1–16. [Online]. Available: http://portal.acm.org/citation.cfm?id=1765931.1765933

[18] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *Proceedings of the 19th annual international symposium on Computer architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 46–57. [Online]. Available: http://doi.acm.org/10.1145/139669.139702

[19] J. Mak and A. Mycroft, "Limits of parallelism using dynamic dependence graphs," in *International Workshop on Dynamic Analysis*, 2009, pp. 42–48.

[20] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *ISCAS'08*, 2008, pp. 1192–1195.

[21] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423

[22] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[23] E. Grochowski and M. Annavaram, "Energy per instruction trends in intel® microprocessors," 2006.

[24] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th annual international symposium on Microarchitecture*, ser. MICRO 25. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 45–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=144953.144998

[25] C. Huang, S. Ravi, A. Raghunathan, and N. K. Jha, "Generation of heterogeneous distributed architectures for memory-intensive applications through high-level synthesis," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 11, pp. 1191–1204, Nov. 2007. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2007.904096

[26] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and t. T. Team, "Scaling to the end of silicon with edge architectures," *Computer*, vol. 37, no. 7, pp. 44–55, Jul. 2004. [Online]. Available: http://dx.doi.org/10.1109/MC.2004.65

[27] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The wavescalar architecture," *ACM Trans. Comput. Syst.*, vol. 25, pp. 4:1–4:54, May 2007. [Online]. Available: http://doi.acm.org/10.1145/1233307.1233308

[28] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore, "Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 1, pp. 62–93, Mar. 2004. [Online]. Available: http://doi.acm.org/10.1145/980152.980156

[29] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 381–394. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.10

[30] M. Budiu, "Spatial computation," Ph.D. dissertation, Carnegie Mellon University, Computer Science Department, December 2003, technical report CMU-CS-03-217. [Online]. Available: http://www.cs.cmu.edu/ mihaib/research/thesis.pdf

[31] G. Venkataramani, T. Bjerregaard, T. Chelcea, and S. C. Goldstein, "Hardware compilation of application-specific memory-access interconnect," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 25, no. 5, pp. 756–771, Nov. 2006. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2006.870411

[32] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan, "Chimps: a high-level compilation flow for hybrid cpu-fpga architectures," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, ser. FPGA '08. New York, NY, USA: ACM, 2008, pp. 261–261. [Online]. Available: http://doi.acm.org/10.1145/1344671.1344720

[33] M. Mishra, T. Callahan, T. Chelcea, G. Venkataramani, S. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM, 2006, pp. 163–174.

[34] G. Venkatesh, J. Sampson, N. Goulding, S. K. Venkata, M. B. Taylor, and S. Swanson, "Qscores: Configurable co-processors to trade dark silicon for energy efficiency in a scalable manner," in *Proceedings of The 44th International Symposium on Microarchitecture*, 2011.

[35] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 4, pp. 441–470, Oct. 2004. [Online]. Available: http://doi.acm.org/10.1145/1027084.1027087

[36] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "Pact xpp&mdash;a self-reconfigurable data processing architecture," *J. Supercomput.*, vol. 26, no. 2, pp. 167–184, Sep. 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1024499601571

[37] J. a. M. P. Cardoso and M. Weinhardt, "Xpp-vc: A c compiler with temporal partitioning for the pact-xpp architecture," in *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, ser. FPL '02. London, UK, UK: Springer-Verlag, 2002, pp. 864–874. [Online]. Available: http://dl.acm.org/citation.cfm?id=647929.740066

[38] H. Gädke and A. Koch, "Accelerating speculative execution in high-level synthesis with cancel tokens," in *Proceedings of the 4th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ser. ARC '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 185–195.

[39] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," *SIGARCH Comput. Archit. News*, vol. 23, pp. 414–425, May 1995. [Online]. Available: http://doi.acm.org/10.1145/225830.224451

[40] P. Yiapanis, D. Rosas-Ham, G. Brown, and M. Luján, "Optimizing software runtime systems for speculative parallelization," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 39:1–39:27, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400698