Exposing ILP in Custom Hardware with a Dataflow Compiler IR

Ali Mustafa Zaidi
Advisor: David Greaves

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

I. Introduction

The difficulty of effectively parallelizing code for multicore processors, combined with the end of threshold voltage scaling has resulted in the problem of *Dark Silicon*, severely limiting performance scaling despite Moore's Law. To address this, not only must we drastically improve the energy efficiency of computation, but due to Amdahl's Law, we must do so without compromising sequential performance.

Designers increasingly utilize custom hardware to dramatically improve both efficiency and performance in increasingly heterogeneous architectures. Unfortunately, while it efficiently accelerates numeric, data-parallel applications, custom hardware often exhibits poor performance on sequential code. Thus complex, power-hungry superscalar processors must still be utilized to improve performance.

The objective of this work is to mitigate the effects of dark silicon by enabling more pervasive utilization of highly efficient custom or reconfigurable hardware for general-purpose computation. 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 (i.e. minimal or no alterations to the code or programming model).

To this end, we develop a new compiler intermediate representation (IR), called the Value State Flow Graph (VSFG), that exposes instruction-level parallelism (ILP) from sequential code even in the presence of complex control-flow. The VSFG is also designed to be directly implementable as custom hardware, replacing the traditionally used Control-Data Flow Graph (CDFG).

To test this new IR, we implement a new high-level synthesis (HLS) tool-chain, that compiles code to the VSFG IR, and then implements it as a Verilog hardware description. Unlike the statically-scheduled execution model of traditional custom hardware, we employ the dynamically-scheduled 'Spatial Computation' model [1], in order to match the dynamic scheduling advantages of out-of-order processors, allowing for better tolerance of variable latencies and statically unpredictable behaviour.

The hardware generated from this tool-chain shows an average speedup of $1.13\times$ over equivalent hardware generated using LegUp, an existing HLS tool. In addition, the VSFG allows us to further trade area & energy for performance through loop unrolling, increasing the average speedup to $1.55\times$, 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.

II. Background and Motivation

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 [2]. Esmaelzadeh et al. identify two main sources of Dark Silicon:

1) *Amdahl's Law*: With the exception of certain numeric, data-parallel applications, most applications in the general-purpose domain lack sufficient *explicit* parallelism to make full use of the ever increasing number of cores on chip [3]. Due to Amdahl's Law, the overall speed-up for such applications is strictly constrained by sequential performance.
2) *Utilization Wall*: Even when an embarrassingly parallel application is not limited by Amdahl's Law, overall performance scaling will still be limited by the Utilization Wall [4]: with each process generation, a decreasing fraction of on-chip transistor resources may be switched at full speed at one time, in order to meet the power budget.

To mitigate the effects of dark silicon, systems architects are increasingly incorporating custom (and reconfigurable) hardware in their designs. Unfortunately, while custom-hardware can improve both efficiency and performance for numeric or data-parallel applications by multiple orders of magnitude [5], general-purpose sequential code often exhibits much lower performance in custom hardware than a typical out-of-order superscalar processor [6], [1], [4].

Such sequential code frequently involves irregular memory accesses and complex control-flow, such as data-dependent branching, function calls & multilevel nested loops. Currently the only means of achieving high performance on such code is through the use of complex, out-of-order superscalar processors that exhibit very poor energy efficiency. Such processors are also not very scalable – energy efficiency worsens exponentially as designers strive for higher sequential performance with increasingly complex architectures [7].

This puts us between a rock and a hard place: without utilizing complex processors, performance scaling is limited by Amdahl's Law and poor sequential performance, but with such processors, the Utilization Wall limits speed-up by limiting the total active resources at any time. Esmaelzadeh et al. [2] focused on desktop, server and workstation domains, that have a reasonable power budget of 20-200W. This problem is exacerbated even further when we consider the increasingly important and rapidly growing portable computing domain, where power budgets are further limited to only 0.5-5W.

A. The Superscalar Performance Advantage

Esmaelzadeh et al. [2] 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 [3]. 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 [8], 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 [9]. 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.

There are two main reasons that out-of-order superscalar processors are able to achieve higher performance on control-flow intensive sequential code [10]:

- 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. In case of branch mis-speculation, processors make use of an in-order commit buffer (or re-order buffer). to selectively commit executed instructions in program order. In case of misprediction, executed instructions from mispredicted paths can simply be discarded from this buffer, preserving correct program state.

Fig. 1 presents a code sample, and Fig. 2a presents its equivalent Control-Data Flow Graph, or CDFG (blue blocks represent '*if*' statement operations, while yellow blocks form the remainder of the *for* loop). Branch prediction in a superscalar processor will be able to overcome the control-flow dependences between the three basic blocks composing the for-loop. Furthermore, branch prediction enables *dynamic* unrollling of the for-loop, and exploitation ILP from across multiple iterations.

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

Fig. 1. Example C Code.

Conversely, custom hardware implementations of sequential code have poor performance primarily due to the lack of effective control-flow speculation mechanisms. Conventional HLS tools would implement the CDFG in Figure 2a directly as custom hardware [11]. 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.

While it is possible to perform speculation in hardware on some *forward* branches through *if-conversion* [12], 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 in spatial hardware without introducing a synchronization bottleneck.

HLS 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 [13], but this can significantly increase the complexity of the centralized finite-state machines, resulting in very long combinational paths that can overwhelm any gains in ILP [14].

Approximating the 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. 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.

For instance, even in the case that the load ("= *A[i]*") instruction from Fig. 2a encounters a cache miss, all the remaining operations that are in flight from the multiple blocks and loop iterations may still execute in dataflow order - only those operations that are dependent on the value of the load will be delayed.

On the other hand, custom hardware typically employs static scheduling – 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.

III. Related Work

The combination of these factors results in custom hardware exhibiting poor performance when implementing general-purpose sequential code via high-level synthesis. Venkatesh et al [4] generate custom hardware *conservation cores* from hot regions of sequential general-purpose code to improve energy efficiency. However, their cores are at best only able to match the performance of an in-order MIPS 24KE processor, while providing $10\times$ better energy-efficiency. Recent updates to their work have improved this to being approx 20% faster than the MIPS baseline [15].

Budiu et al. attempt to improve performance by implementing dynamically scheduled execution in the form of asynchronous static-dataflow hardware generated from C code [1]. While the generated hardware manages around $100\times$ better energy efficiency than a simulated 4-wide out-of-order superscalar processor, it still proves to be 30% slower on average [6].

On the other hand, Govindaraju et al's DySER architecture 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 to about 9% when implementing sequential code [16].

IV. Matching Superscalar Performance with the Value State Flow Graph

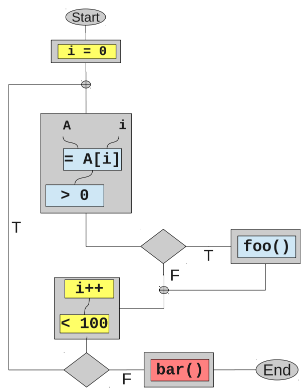
To overcome the sequential performance issues of custom hardware, we propose two key changes during high-level synthesis:

- Instead of using a static scheduling based execution model for custom hardware, a dynamically scheduled execution model like *Spatial Computation* should be used [1], 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.

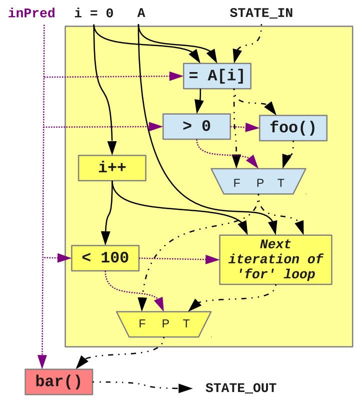
A. The Value State Flow Graph

To address the sequential code performance limitations imposed by control-flow in custom hardware, we have developed a new compiler IR for implementing spatial computation, called the Value State Flow Graph (VSFG), to be used as a replacement for the CDFG during high-level synthesis. The VSFG is based on the VSDG but modified for direct implementation in hardware as a static-dataflow machine.

The VSFG for the code from Fig. 1 is shown in Fig. 2b. Unlike the equivalent CDFG (Fig. 2a), the VSFG is no subdivision of operations into basic blocks, and consequently, no notion of *flow of control* from one block to another. Instead, only dataflow dependences are



(a) The CDFG for the code from Fig. 1.



(b) The VSFG for the code from Fig. 1.

Fig. 2. The equivalent CDFG and VSFG for the code given in Fig. 1.

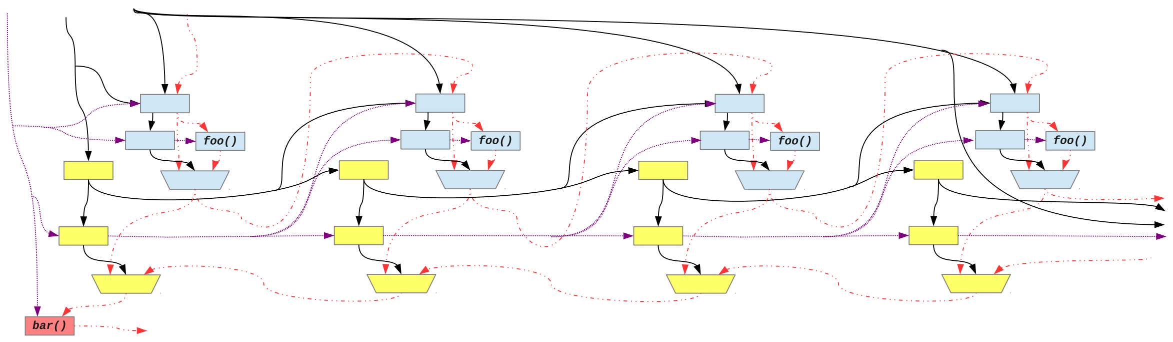


Fig. 3. The VSFG from Fig. 2b with the loop unrolled 4 times.

represented, along with a sequentializing state-edge (dashed line in Fig. 2b) to ensure that all side-effecting operations occur in the correct program order. Instead of a flow of control between basic blocks, execution of operations is controlled through the use of *predicates* (purple dotted lines in Fig. 2b): boolean expressions generated based on the control-flow of the original CDFG.

The VSFG is a hierarchical graph – all loops and function calls are represented as nested subgraphs. From the perspective of any 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 long and unpredictable execution latencies. As with other dataflow operations, multiple such nested subgraph operations may execute concurrently, so long as their dataflow dependences are satisfied, as opposed to the CDFG, where only operations from within a single active basic block are allowed to execute concurrently.

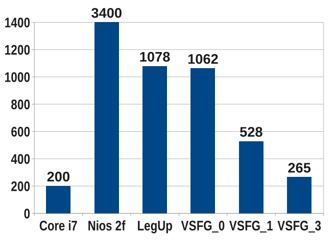
The VSFG is a directed acyclic graph – loops are implemented without introducing explicit cycles in the graph by representing them instead as tail-recursive functions. As can be seen from Fig. 2b, the *"Next iteration of the for-loop"* is represented as a nested subgraph in the same way as both the *foo()* and *bar()* functions.

B. Advantages of the Value State Flow Graph

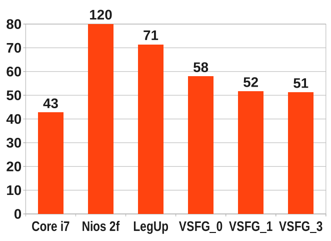
The acyclic, tail-recursive representation of loops presents a key advantage when we try to extract ILP from across multiple iterations of a loop. Any of the nested subgraphs in a VSFG can be *flattened* into the body of the parent graph. In the case of loops, flattening the subgraph representing the tail-recursive loop call is essentially equivalent to unrolling the loop. Fig. 3 shows the *for*-loop from Fig. 1 *unrolled* 4 times. 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 by unrolling each loop within a loop nest independently of the other loops. (Note that in the actual hardware implementation, cycles must be reintroduced once the appropriate degree of unrolling has been done for each loop).

Thanks to the explicit control predicates, is also possible to perform aggressive control flow speculation by selectively controlling the execution of subgraphs. For instance, for the *foo()* function subgraph, we may choose whether this function executes speculatively or not: the predicate input to *foo()* may be used to only allow its execution when the predicate is true, thereby providing no speculation. Alternatively, the function may start executing irrespective of the predicate value. In this case, the predicate value will be passed into the subgraph, where side-effect free operations may execute speculatively even before the predicate value becomes available (all side-effecting operations will always be predicated).

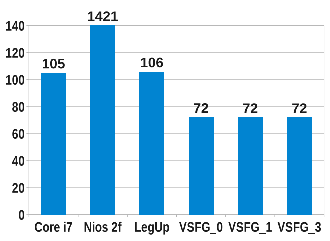
Another advantage of having control flow converted in to boolean predicate expressions is the ability to perform *control dependence analysis* to identify regions of code that are control-flow equivalent and may therefore non-speculatively execute in parallel. Consider the *bar()* function in the CDFG (Fig. 2a). Despite aggressive branch prediction, a superscalar processor will not be able to start executing *bar()* until control exits the *for* loop. Similarly, when the *if* branch is predicted-taken, the superscalar processor must switch from executing multiple dynamically unrolled copies of the 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].



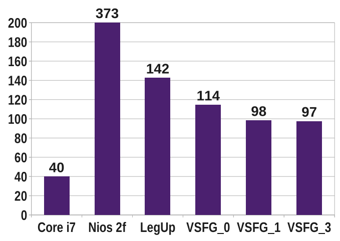
(a) epic (×1K cycles)



(b) adpcm (×1K cycles)

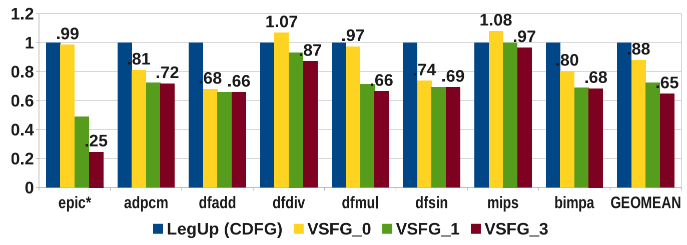


(c) dfsin (×1K cycles)

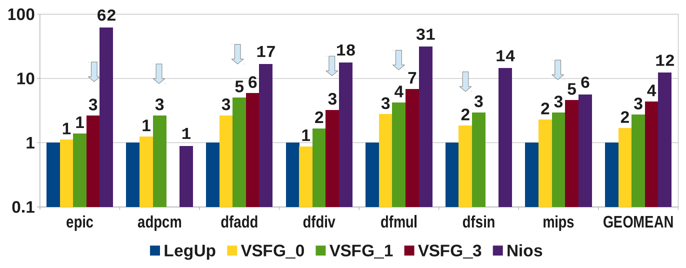


(d) bimpa (×1M cycles)

Fig. 5. Performance Comparison (Cycle Count) vs an out-of-order Intel Nehalem Core i7 processor, and an Alteral Nios IIf in-order processor.



(a) Performance in Cycle Counts vs LegUp.



(b) Energy Consumption comparison vs LegUp and an Altera Nios IIf in-order Processor.

Fig. 4. Performance and Energy Comparison of VSFG, normalized to LegUp results.

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, it becomes possible in Fig. 3 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].

V. Evaluation Methodology & Results

To evaluate the energy and performance potential of the VSFG IR, we implemented a HLS toolchain to compile from the LLVM IR to static-dataflow custom hardware. we present results for 3 versions of VSFG hardware: VSFG_0 has no loop unrolling/flattening, VSFG_1 has all loops unrolled once, and VSFG_3 has all loops unrolled thrice. We compare these results against LegUp [11], an established HLS tool that utilizes the CDFG IR and static execution scheduling (Fig. 4). All generated circuits were targeted for implementation on an Altera Stratix IV GX FPGA.

We also compare cycle-count[1] performance against two conventional processors: an Intel Nehalem Core i7 Processor – simulated using the Sniper interval simulator [19] – as well as an Altera Nios II/f soft-processor (Fig. 5). Both processors are configured with perfect L1 caches and a hit latency of 1 cycle.

Six of the benchmarks used are part of the CHStone benchmark suite [20], while two others are home-grown: *bimpa* is a memory and control-flow intensive neural-net simulator, and *epic* is a matrix transpose function – both of the latter were selected specifically because they exhibit complex, data-dependent control flow.

The generated VSFG hardware achieves a geometric mean speedup of 1.55× (max 4.05×) over equivalent hardware generated by LegUp (Fig. 4a), and is able to better approach (in some cases even exceed) the cycle-count performance of the Intel Core i7 (Fig. 5). While this performance incurs an average 3× higher energy cost than LegUp, the VSFG-based hardware's energy dissipation is still only 0.25× that of a highly optimized in-order Nios IIf processor (Fig. 4b).

It is important to note that these results have been generated from a largely unoptimized implementation of our toolchain. Further optimizations that incorporate alias-analysis based memory disambiguation and parallelization, as well as careful management of the degree of speculation in the VSFG-based hardware, are expected to significantly improve both performance and energy efficiency even further.

VI. Contributions

In this research, our goal is to mitigate the effects of dark silicon by improving sequential code performance in custom and reconfigurable hardware. To this end, this work makes the following contributions:

- A new compiler intermediate representation (IR), called the VSFG is developed, that exposes ILP from sequential code even in the presence of complex control flow.
- To test our new IR, a new HLS tool is implemented that compiles high-level language code to VSFG-based static-dataflow custom hardware.
- We demonstrate that the generated custom hardware (a) provides better performance than hardware generated by LegUp, an established HLS tool, and (b) is able to approach (in some cases even exceed) the cycle-counts of an Intel Nehalem Core i7 processor. While this performance incurs an average 3× higher energy cost than LegUp, the VSFG-based hardware's energy dissipation is still only 0.25× that of a highly optimized, in-order Altera Nios IIf processor.
- We provide recommendations for how both the energy efficiency and performance of our hardware may be further

[1]Cycle-counts provide a means of comparing the ILP exposing potential of the various implementations, while allowing us to temporarily side-step implementation substrate and tool-chain optimization issues. This is especially important given the 20× difference in the operating frequency of FPGA-based circuits (running on average at 150MHz) and a full custom processor (running at 3GHz).

improved by implementing simple compiler optimizations, such as performing alias-analysis to partition and parallelize memory accesses, as well as how to reduce the energy overheads of speculation.

We believe that our new VSFG IR, coupled with the dynamically scheduled, static-dataflow execution model is a promising step towards addressing the problem of dark silicon by facilitating the development of high-performance custom hardware.

In the future, with the incorporation of alias-analysis to parallelize memory accesses in the VSFG, it may even be possible to overcome the ILP Wall faced by superscalar processors and exploit much greater ILP from control-intensive sequential code [18].

References

[1] 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

[2] 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

[3] 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

[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] 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

[6] 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.

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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

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

[14] 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

[15] 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.

[16] 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.

[17] A. C. Lawrence, "Optimizing compilation with the Value State Dependence Graph," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-705, Dec. 2007. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-705.pdf

[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] 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.

[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.