Exploring the Parallel Programming Design Space of Proximate, a Multi-Tile Programmable Accelerator

(CS 758 Project Report)

Vinay Gangadhar and Khai Zhao
University of Wisconsin-Madison
{vinay, kzhao32}@cs.wisc.edu

Abstract

The slowing of Moores law and Dennard scaling is limiting the performance improvements of single core processors. Increasing clock frequency any farther will lead to high leakage current and infeasible power consumption. Further, these cores are extra-ordinarily inefficeint and are optimzied only for single thread arbitrary programs. Over the past decade, focus has been shifted to multicore processors to increase throughput by having multiple cores to target different types of parallelism - instructoion (ILP), data (DLP) and thread (TLP). However, even the multi-core processors are not scalable for parallelization beyond a point due to Amdhal's law. To address the challenges of rising dark silicon and the end of Dennard scaling, architects have turned to heteregenous architectures with special purpose domain specific accelerators (DSAs), for higher performance and energy effieincy. While providing huge benefits, DSAs are prone to obsoletion due to domain volatility, have recurring design and verification costs, and have large area footprints when multiple DSAs are required in a single device to reap out the benefits of different applcaition acceleration. To attack such problems of DSAs, while still having

Because of the benefits of generality, this work explores how far a programmable architecture can be pushed, and whether it can come close to the performance, energy, and area efficiency of a DSA-based approach. Our insight is that DSAs employ common specialization principles for concurrency, computation, communication, data-reuse and coordination, and that these same principles can be exploited in a programmable architecture using a composition of known microarchitectural mechanisms. Specifically, we propose and study an architecture called LSSD, which is composed of many low-power and tiny cores, each having a configurable spatial architecture, s

With the traditional scaling laws Dennards and Moores law slowing down in recent years, a special class of accelerators called Domain Specific Accelerators (DSAs) are being explored to reap performance and energy efficiency for particular embedded application domains. Though DSAs obtain 10x to 1000x performance and energy benefits compared to a general purpose processor, they compromise programmability and are prone to obsoletion due to domain volatility, and also incur high recurring design and verification costs. This necessitates the need for another class of accelerators called Pro-

grammable Accelerators which retain the programmability for different application domains and try to achieve the performance, area and energy efficiency of each DSA. This project focuses on building one such programmable accelerator for neural network domain. We mainly build upon the prior work and preliminary modeling results done for this specialization engine based on specialization principles it employs1 . For this course project, we targeted deep neural network domain and built a specialization engine called Programmable Engine for Neural Networks (PENN). It should be able to execute variety of deep neural network applications like convolution, classification and pooling kernels. The project focuses on: i) Exploring the trade-offs of many fine-grain design decisions for PENN based on a thorough analysis of neural network workloads, ii) End-to-end functional and timing model implementation of PENN using CHISEL [3], iii) Building complete software toolchain to compile the programs and configure PENN. We have evaluated our design for power and area with state-of-the art neural network DSA DianNao [5]. We have limited our study only to deep neural networks for this project

1 Introduction

The end of classical device scaling means that the power per unit area on chip is rising with each technology generation. This implies that architectures for future technology nodes will not be able to power-on all components of the chip simultaneously, with some estimates being 50% "dark silicon" by 8nm [16], which is less than 10 years away. This trend, the utilization wall, will curtail expected performance improvements. Interestingly, much of the core's energy is not expended in the functional units, but rather in the power-hungry structures needed for attaining reasonable performance on general purpose workloads. To exploit this, architects have in part turned to hardware specialization and accelerators, which sacrifice generality for efficiency in executing either specific computations or computations for certain application domains.

Though accelerators hold great promise, with many new accelerator designs showing significant efficiency and performance gains, the research and insights have been fragmented. Newly proposed accelerators are evaluated in their own toolchain, generally with a specific general purpose core, and with a set of chosen benchmarks. Solving this by integrating tens of accelerators into a single simulation system and developing compatible compilers for them is intractable due to development time. This means that attaining insights into accelerators which transcends specific simulators, core design points, evaluation metrics and applications is extremely difficult. This limitation hinders our ability to understand the future of acceleration in terms of their behavior, design and use.

Going forward, the architecture community needs a methodology for modeling acceleration which is detailed and accurate, yet simple and abstract, to help unify and improve the current understanding. With such a tool, we can begin to address the biggest challenges of acceleration: First, we must understand

where the biggest benefits of acceleration could come from, and what are the biggest limitations, so we can best focus our efforts on problems which will push the limits of their effectiveness (*Finding Accelerator Limits/Opportunities*). Second, we need a strategy for designing accelerators which can explore a broad range of the possible space in a short amount of time (*Effective Practices for Accelerator Design*). Finally, we need practical and flexible ways to employ multiple accelerators at runtime so that the promises of accelerators can be achieved in a wide variety of domains (*Enabling Practical Multi-Acceleration*).

In pursuit of these challenges, this dissertation proposes a novel abstraction called the Transformable Dependence Graph (TDG), which can model certain important forms of acceleration. It is designed to accurately capture interactions between the accelerator, general purpose core, and application at sub-instruction granularity. The core idea is to represent the programs execution trace as a dependence graph of micro-architectural events, and to perform transformations on this representation to model various forms of acceleration. This concept is based on the dependence-graph of Fields *et al.* [21, 20], and our contribution is in showing how graph transformations can model acceleration.

1.1 Completed Work

Completed work has shown that our model and implementation is accurate in capturing both the out-of-order execution of processors, as well as the transformation from OOO dependence-graph trace to model four different classes of accelerators. Our results have shown that, using the dependence-graph model, we can achieve an average error of less than 15% in both performance and energy reduction for all classes of accelerators we target. In general, there are certain accelerator classes which can be modeled accurately, discussed in detail in Section 5.7 (Page 37). The flexibility, accuracy, and high-abstraction of the TDG is useful for solving a variety of problems beyond just modeling, and forms the basis for the proposed research.

1.2 Summary of Proposed Work

Figure 1 gives a high-level overview of each proposed work compared to the current approaches for each problem, and they are outlined below.

- **Limits and Opportunities of Acceleration:** For the future of acceleration, it is important both to understand the fundamental limitations of accelerators, as well as to know where the largest benefits can come from. We can better understand accelerators by modeling hypothetical designs which break current limitations, and observing their new behavior. Current approaches using simulators would be too tedious because they require ad-hoc accelerator implementation and integration with multiple simulators. The proposed approach employs the TDG to simplify modeling and unite the modeling of

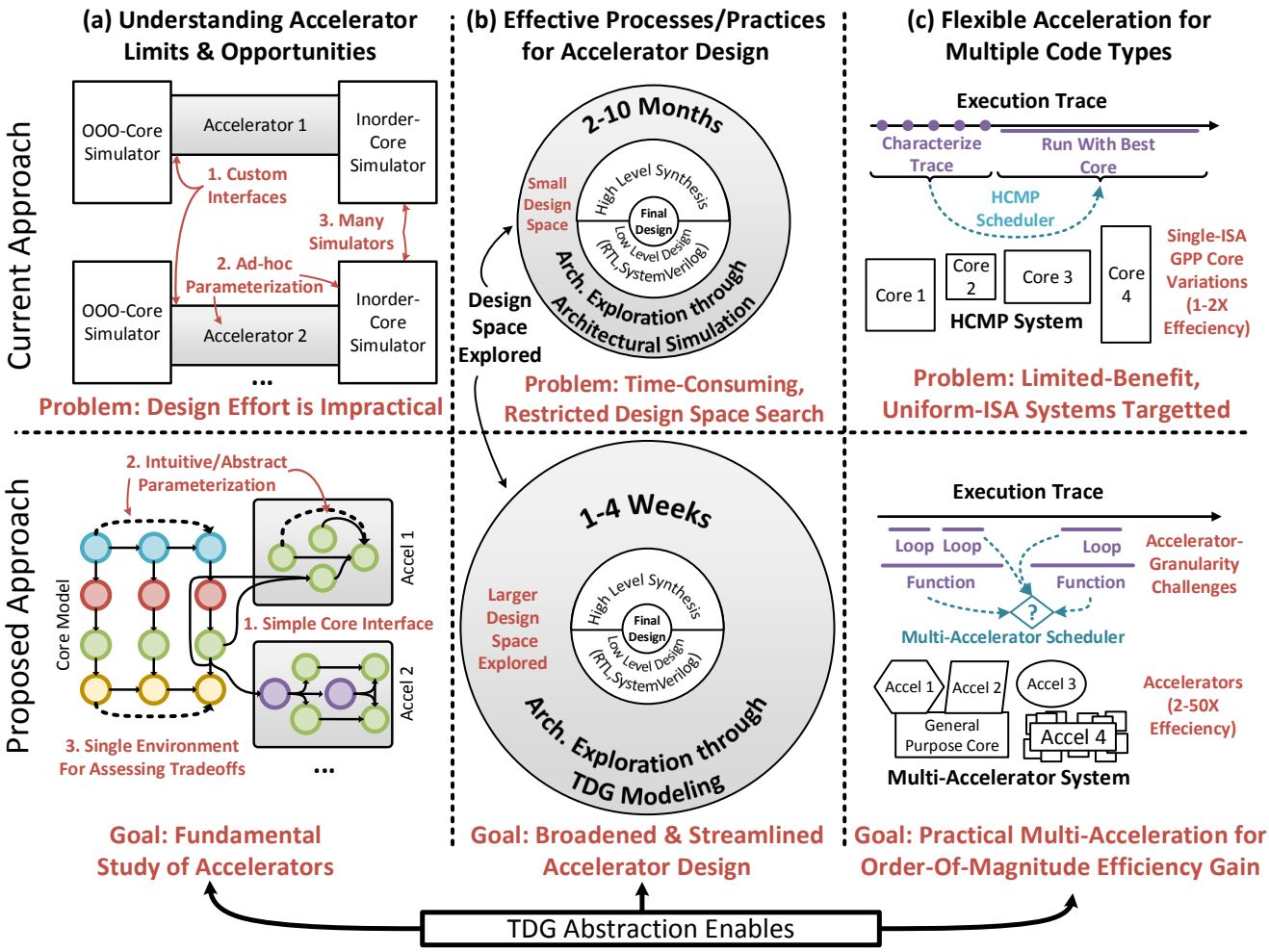


Figure 1: Summary of proposed work as compared with current best techniques.

accelerators under a common framework, making the study of their nature consistent and practical. This work's goal is to uncover the fundamental limitations of accelerators, to show what are the most important challenges for future accelerator architects.

- **Effective Processes/Practices for Accelerator Design:** Existing accelerator design strategies are ineffective because they rely on tedious simulator modeling, consider accelerators in isolation, and inappropriately bin applications to accelerator designs. Overall, they are able to cover less of the design space, and require more design time. To address this, this research proposes an accelerator design strategy which leverages the TDG. Primarily, it presents a higher level of abstraction, both broadening the explored design space while hastening the process. The unified framework makes it simple to compare with existing accelerators during the design-space exploration, which can help prune the design space quickly. Also, many TDG models can be used together to find the regions with the biggest potential for new accelerators, giving insight to architects without the need to bin accelerators. The research so far shows how to use the TDG to help design an accelerator for nested loops, a

contribution on its own.

- **Enabling Flexible/Practical Multi-Acceleration:** By employing multiple accelerators in a single system, it is possible to achieve flexible efficiency for a variety of code types. Current approaches lack the ability to target systems more general than uniform-ISA heterogeneous architectures. To attain the benefits of multi-acceleration, it is necessary to partition regions of applications onto the most beneficial accelerator. This decision can be complicated, because accelerators are only legal on different granularities of code. Also, certain accelerators can have serious drawbacks if applied on the wrong application regions. The proposed research is to explore static and dynamic mechanisms, including intuition-guided static approaches, compiler-generated TDG-based prediction, multi-granularity sampling techniques, and machine-learning based performance/energy models.

1.3 Relation to My Previous Work

The inspiration for this dissertation topic stems from my previous work in creating a unifying general framework for spatial architecture scheduling [46], which received a distinguished paper award. That work leveraged a mathematical theory, namely integer linear programming, to create abstractions which enabled the modeling of scheduling constraints and characteristics common across a wide variety of architectures. This work in turn inspired us to co-author a synthesis lecture on the use of optimization and mathematical modeling in computer architecture [45].

My previous work is related to this dissertation work in two ways. First, the principles of creating a general framework, using higher-level abstractions and mathematical modeling of architectural phenomenon are common to both. Second, and more concretely, the instruction scheduling techniques developed are modified and employed for some newly developed accelerators we consider.

1.4 Document Overview

Section 5 describes completed work in using the TDG to model various forms of acceleration, in terms of modeling and validation. The next three sections describe the proposed work: Section 4 for studying the limits and opportunities, Section 3 for using the TDG to aid the design of new accelerators and Section 2 for enabling practical multi-acceleration. Each section of proposed work contains a *motivation* section, a *research approach* section to explain how we address the problem, a *preliminary results* section, a *proposed work* section which overviews the remaining work to be done, and ends with *related work*. Section 6 summarizes the work, Section **??** outlines deliverables, and Section **??** describes the proposed dissertation schedule.

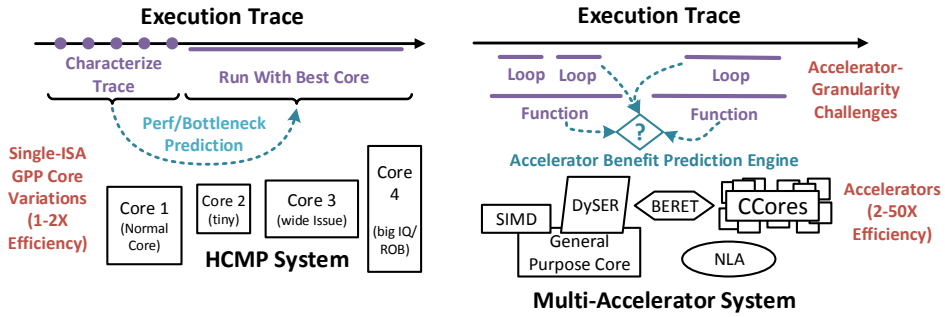


Figure 2: Current versus proposed approach for Multi-Accelerator scheduling.

2 Enabling Practical Multi-Acceleration

2.1 Motivation

Accelerators are designed to provide fundamentally different energy-efficiency and performance tradeoffs, depending on the properties of the application. Actually achieving the potential speedup of multi-acceleration requires practical techniques which can select the correct accelerator for the appropriate region of code. The current research thus far, outlined in Figure 2, has only addressed the scheduling problem across uniform-ISA core types [43, 39], which can only achieve modest efficiency gains. These techniques are not directly applicable to accelerators, as their tradeoffs are more distinct, and their limitations are more restrictive.

1. **Static versus Dynamic:** The first challenge is in whether the accelerator scheduling should be static or dynamic. More specifically, should a compilation-time pass decide which regions should be mapped to which accelerators, or should a runtime phase somehow profile the dynamic execution and make decisions based on that. Static algorithms can have a much higher runtime, but dynamic approaches can take advantage of additional information. The correct answer to this question will depend on whether there is a significant advantage in dynamism over time given a particular scheduling region. Also, dynamic approaches cannot be used to decide whether baked-in accelerators like Conservation Cores should be built in the first place, where as static-schedulers could.

2. **Granularity of Scheduling:** The second challenge is that different accelerators operate on different granularities, and hence overlap in their scheduling regions. For example the BERET architecture only targets traces of inner loops, while Conservation Cores targets entire functions containing potentially many loops, or can also target just inner loops. Since the decision of one region affects the other, greedy decisions can lead to sub-optimal solutions.

3. **Evaluation Metrics:** Both energy-efficiency and performance are important metrics depending on

the setting, which could even change over time for a given device. For instance, a phone may want to switch from optimizing for performance to energy when the battery is low. Techniques which can provide flexibility would be ideal.

4. **Harmfulness of an Incorrect Decision:** Finally, in choosing a certain accelerator for program region, an ideal solution would guarantee that at a minimum, whatever performance or energy-efficiency is provided by the general purpose processor is preserved by the accelerator. However, it is possible that accelerators can degrade either metric. Care must be taken not to do more harm than good.

2.2 Research Approach

Before designing techniques for enabling multi-acceleration, we first explore whether there is inherent value. Also, as it is an unexplored topic, the way forward in integrated, automatic multi-accelerator scheduling is unclear. Therefore, it is prudent to develop and compare a variety of strategies which have different tradeoffs. The remainder of this section describes four initial approaches which will be investiaged, and also describes a mechanism for handling multi-granularity which is common across techniques.

Value of Multi-Acceleration Results from section 4 provided quantitative evidence suggesting there exists significant energy-efficiency and performance benefit from utilizing different accelerators on different benchmarks. Going further, Figure 3 shows the benefits of multi-acceleration for both the inorder and OOO core. On the bottom half of each graph is the percentage of time each accelerator is the most beneficial in terms of performance. There are several useful observations here, first, structured hardware accelerators like Conservation Cores and BERET are much more utilized on the inorder core. More importantly, even within a single application, depending on the region being accelerated, different regions favor different accelerators, some even using all five(including no-accelerator) design points. On the top half of the graph is amount that each benchmark is sped up over choosing just one accelerator for that benchmark. For some benchmarks, multi-acceleration at a fine granularity can have up to 50% performance improvement. We expect these numbers only to increase with further accelerator development.

Program Structure Tree As mentioned earlier, scheduling is made difficult because the choice to schedule at one granularity could preclude a beneficial scheduling at a finer or coarser granularity. This problem is easily solvable using a greedy algorithm if we have perfect information of how much benefit each accelerator provides at each scheduling granularity. To do this, we require a structure called the program structure tree (PST). It is highly related to the call graph, except that it also includes nodes for loops and traces through

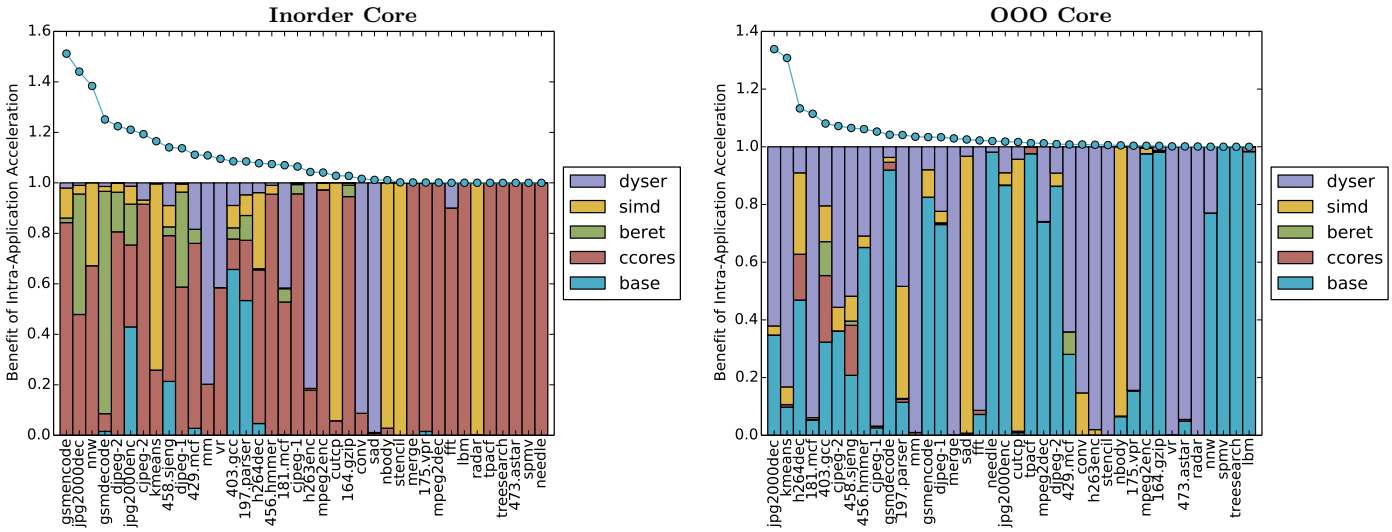


Figure 3: Benefits of Fine-Grain Multi-Accelerator Scheduling

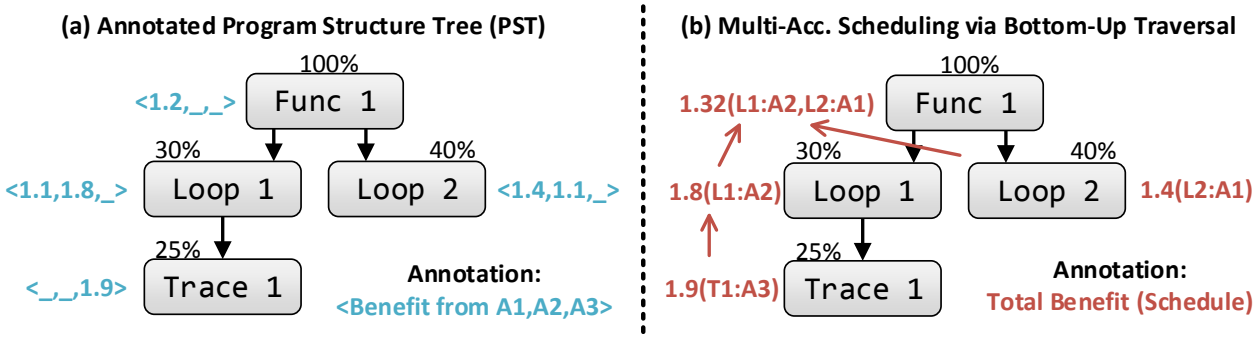


Figure 4: Scheduling via the Program Structure Tree (PST)

loops that can be targeted by certain accelerators, and it also doesn't naturally handle recursion. This is acceptable for our use case since none of the accelerators we model can utilize recursion.

Scheduling can be performed on the PST as we describe next. Figure 4(a) shows an example PST which is annotated with the percentage of execution time each region is responsible for, along with a "benefit," vector, where each entry represents the amount of benefit each accelerator can provide over the general purpose processor in terms of a certain metric. In this example we assume performance is the metric of interest. Figure 4(b) shows how to traverse the tree from bottom up to compute the schedule. At each node in the tree, a decision needs to be made whether an acceleration at the next lower level, or the current level is better. For performance calculation we can apply Ahmdal's law. For example, "Loop 1," must choose whether accelerating at "Trace 1" or at it's own granularity is better. Using Ahmdal's law, using "Trace 1" inside Loop 1 would yield a speedup of: $1/((1-\frac{.25}{.3})+.\frac{.25}{.3}/1.9) = 1.65\times$ performance. This is worse than the best acceleration at granularity "Loop 1," so "Loop 1" instead selects accelerator 2 (A2) at it's own

granularity. This continues until the top of the tree, where all scheduling decisions would be made.

Of course, attaining the information necessary for using this strategy is the more challenging aspect of multi-granularity scheduling. Below we describe four techniques, the first uses an adhoc technique which does not require the PST, and the other three techniques utilize the PST.

Static Technique: Intuition-Guided Scheduling Perhaps the most straightforward method of scheduling is to use heuristics to decide if a region should be scheduled to a particular accelerator. For example, if an inner loop has independent iterations, and the communication cost is low, the DySER architecture might be the best choice. Else, if the DySER communication cost is high, choose SIMD. This is the only scheduling technique we propose that does not use the PST for scheduling, and as such, it can miss out on acceleration opportunities from lack of multi-granular scheduling.

Static Technique: TDG-Based Prediction An alternate static approach would be to re-purpose the TDG as a prediction device. Instead of generating a dependence graph from the binary trace as we do now, we could build the graph for different accelerators based on the compiler's intermediate representation. Of course, dynamic latencies (memory latency, misprediction latency), and program inputs (to determine loop iteration/function counts) would have to be estimated, or Monte-Carlo (randomization) techniques could be employed. The predictions, and estimated region time would be fed to a PST traversal for scheduling.

Dynamic Technique: Mixed-Granularity Sampling The most natural dynamic approach would be a sampling based one. When the GPP reaches an acceleratable region, it could try each legal accelerator for that region in turn for some short number of cycles. Sampling would have to occur at multiple granularities, and calculating the best choice would entail a PST traversal at run-time per scheduling decision. After attaining samples and making a decision, it could run the best accelerator for some longer period of time before recalculating. The primary overhead here is in the many samples which would be required for the PST traversal, where many inopportune accelerators would have to be temporarily applied.

Dynamic Technique: Machine Learning As other researchers have discovered, the performance of an accelerator may be able to be predicted by machine learning techniques which interpret properties of the general purpose processor's execution. One approach would be to use performance counters as an input to a regression model to predict the metric of interest. Note that this technique obviates the need to actually run on the accelerator prior to making a decision, so should incur much less frequent switching and overall overhead. However, the overhead of PST traversal would still be required. Linear regression is one viable candidate for a machine learning technique as it can be computed extremely efficiently.

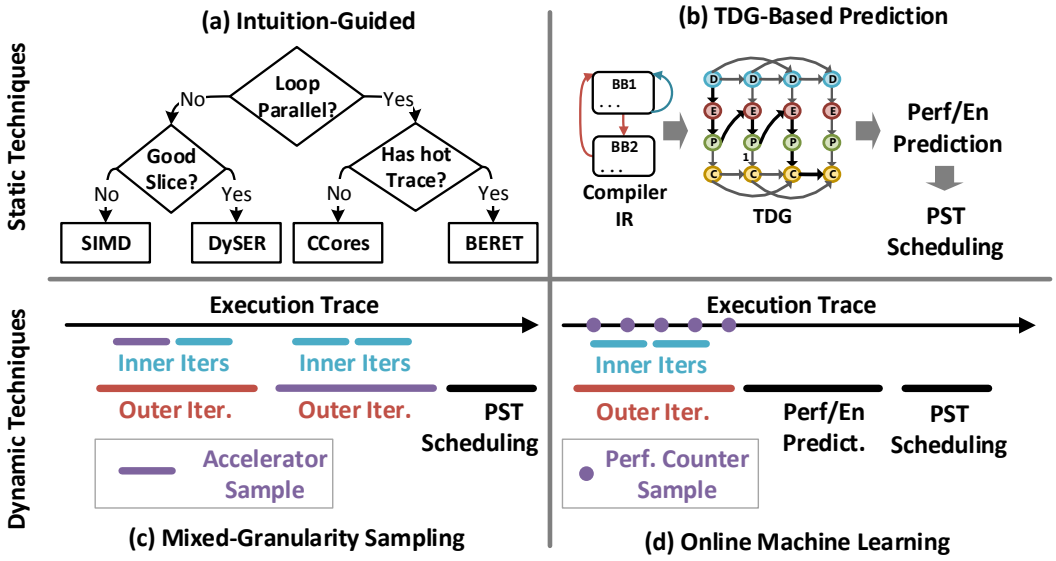


Figure 5: Potential Scheduling Techniques

2.3 Proposed Work

Static vs. Dynamic Scheduling Analysis We will first investigate whether we should immediately rule out either static or dynamic scheduling. We can do this by determining the best dynamic schedule for a particular application by running the application with the current set of models, and recording the cycles during region transitions. We can then compare the best acceleration where the acceleration is always the same as one where we allow the accelerator to change across instantiations of the region.

Scheduler Implementations The next task is to create a version of the TDG which is multi-acceleratable. Currently, only one acceleration transformation can be applied to an entire program. Further implementation will allow the graph to accelerate different portions with different accelerators. After this, depending on what we decided about static/dynamic scheduling, we will implement the proposed scheduling techniques.

Scheduler Evaluation For evaluation, we will target a variety of accelerator compositions, including the most challenging composition which incorporates all accelerators considered so far, even the nested loop accelerator (NLA) described in the previous section. We will compare both how much raw benefit they can provide in performance and energy efficiency, and also how flexible they are in targeting different goals (speedup, energy efficiency, or some combination). The larger question we hope to answer is whether or not multi-acceleration is practically achievable, and how much benefit is likely to come from it.

2.4 Related Work

Heterogeneous scheduling has been addressed before in different settings. One example is work which uses FabScalar to create heterogeneous single-ISA multicore [43]. The technique they use is to treat one core as the baseline core, and switch to "accelerator" cores to alleviate certain bottlenecks in applications. While conceptually similar, this related work targets processors with only minor relative performance benefits, does not deal with the mixed-granularity problem, and does not face significant risk from incorrect choice.

Another example is Composite Cores, which uses two different frontend pipeline designs to feed the same back-end [39]. One front end is high-performance and out-of-order, while the other is in-order and energy-efficient. In their scheduling paper, they show how to use trace based prediction mechanisms to out-predict simpler sampling techniques [50]. Our work differs substantially because we are targeting accelerators that go far beyond merely changing the microarchitecture for a ISA. The proposed work targets fundamentally distinct architectures which can provide potentially an order of magnitude further energy-delay improvement.

3 Effective Practices for Accelerator Design

3.1 Motivation

The accelerator design space is vast. With each new year, many accelerators are put into products, and even more are proposed through research. Table 1 highlights just a few of the commercial and research accelerators which exist today, in several categories, based around the program properties they exploit and their hardware mechanisms. Our research so far has shown how to take a seemingly impossible task – comparing and evaluating accelerators under a common framework – and made it tractable by providing a new abstraction. However, even implementing a TDG-based model for every accelerator available would be ineffective, due to the effort of testing, validation, etc. Going beyond the questions of which accelerators are useful, and how do they compare, is perhaps a more important question: what accelerators would we design for the next generation of computer architecture?

We broadly outline the impact of each phase of the current accelerator design process in Figure 6, where the area of each circular slice indicates the amount of design-space covered in that phase. To give more detail, architects first attain insight into possible accelerator designs using application specific knowledge or analysis of the inefficiency of general purpose processors. Analytical models can be employed, but are generally ineffective because they lack the ability to comprehend application-accelerator interactions. Following a high level design proposal, simulators or simulator extensions are designed and built, while concurrently either custom microbenchmarks are developed or new compilers are developed. Wit some initial version of

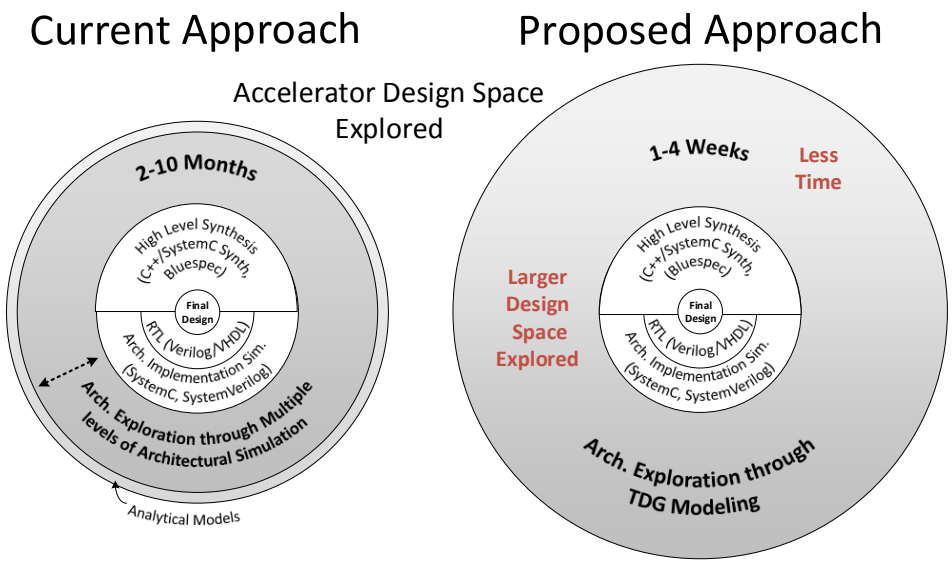


Figure 6: Current versus proposed approach for Accelerator Design Practices.

each, the design proceeds by modifying the newly developed simulators and their compilers and running repeated simulations. After a design is chosen, the implementation phase begins, either using high-level synthesis tools or lower level RTL. Only minor changes in the design can be realistically performed at this point. This entire process is repeated for each potential accelerator design, usually in parallel by separate teams of architects.

While the above may seem like a natural and plausible strategy, there are at least three potential pitfalls.

- **Inappropriate Application Binning:** Under the described framework, choosing the correct set of application domains to target with accelerators is very difficult. It becomes easy to *over-design* by binning applications too aggressively and creating a set of accelerators which does not provide much benefit over a more conservative set. It is also easy to *under-design* by binning too coarsely, and missing out on potential opportunities.

- **Tedious Design Process:** Designing and implementing simulators and compilers for each proposed design, along with the effort of exploring the design space in a simulator model can be tedious. Part of the problem is that simulators require a very low-level view of architecture design.

- **Ignorance of Benefit Overlap:** Since accelerators are usually designed in isolation from each other, architects remain unaware of the overlapping benefits of designs. Since only one accelerator is active at a time, overlap in benefit provided is equivalent to wasted designer effort.

Overall these pitfalls have the cumulative effect that the design time for the accelerator is too high, and

Acc-Class	Properties Exploited	Hardware Strategy	Examples
Structured Hardware	Low-DLP, Mem. Bound Code	Simplified Execution & Offloading	**Conservation Cores** [56], **BERET** [30], OptimoDE [13]
SIMD	Regular Data Parallel Code	Parallel Datapath/Vec-Mem Interf.	**SSE**, SODA [38], VEAL [11], Larrabee [52], XEON PHI [9]
CGRAs	Computation Pattern Re-use	Reconfigurable Datapaths	**DySER** [28], PIPERENCH [25], CCA [12], FCC [23]
Domain Specific	Application Specific Behavior	Custom Datapaths & Memory Interfaces	HARP [61], Convolution Engine [51], **NPU** [17], H.264 Decoding [31]

Table 1: Accelerator Classes & Application/Core Interaction

the design space covered overall is much less. This research proposes a design strategy, based on modeling with the TDG, to aid the early stage design process for accelerators. Its aim is to address the above pitfalls of conventional design, hastening and broadening design-space exploration, and to eliminate (or reduce) the need for simulator-based design-space exploration. This is shown graphically in Figure 6. This section first describes our methodology, then present preliminary results in applying this strategy in designing a particular accelerator. Both the strategy and particular designs will be contributions of this work.

3.2 Research Approach

The accelerator design strategy we propose contains four phases, and we describe it below, highlighting how it addresses the pitfalls described earlier. It bears strong similarities to the existing practices, which we see as a strong point to its successful adoption.

1. **Analysis of Unnaccelerated Regions** The first step is to use existing models to automatically find regions which have "untapped" potential, and use intuition to find patterns or features of regions which could be exploited. That our tool guides architects to unexplored application domains helps eliminate the improper application-binning pitfall. (Section 3.3.1)

2. **High-Level Design Proposal** Using insight from first step, the second step is to propose a high-level design which can take advantage of exploitable regions. Where a conventional high-level design would be defined merely "conceptually" in a typical design process, using our approach, it can be defined rigorously by creating a TDG transformation which captures the most essential facets of the architecture. Since our approach avoids describing accelerators with ad-hoc simulator models, it helps to reduce the pitfall of tediousness of design. (Section 3.3.2)

3. **Design Space Exploration** After enumerating possible design choices within the high-level design, the third step is performing design space exploration using TDG modeling to find the simplest design that meets performance/energy targets. Because our methodology allows us to keep multiple accelerator models in the same evaluation framework, it remains easy to determine if accelerators are providing

overlapping benefit, which helps prevent this pitfall. Also, the analysis or "meta-information" gathering component to the TDG acts as an optimistic compiler, allowing the designer to separate software and hardware concerns. (Section 3.3.3)

4. **Design Refinement** The final step is to iteratively refine the design by developing concrete mechanisms and update the TDG model. The tediousness of design is also reduced in this stage because of the employment of high-level models which can be iteratively added to. (Currently Proposed Work)

3.3 Preliminary Results

In this section, we describe the process and benefits of our proposed design strategy in more detail by applying it to a specific design. Both the specific designs which we deliver, as well as the design process, will be contributions of this proposed work.

3.3.1 Analysis of Unnaccelerated Regions

Using TDG models, it becomes possible to analyze a program in a region-by-region fashion, to find those regions which cannot be accelerated by the existing models, or which have significant potential. We use the methodology below to analyze these regions, and then we determine the reasons which they are not accelerated.

Methodology The regions we consider, from smallest to largest granularity, are traces, inner loops, outer loops, and whole functions. To be clear, every cycle of the program is accounted for by some region. We run our models on the benchmarks described in the previous section (SPEC,Mediabench,Parboil,Intel TPT), and with the DySER, SIMD, BERET, and CCores models. For the purposes of this study, we consider a region to be "unaccelerated" when the degree of speedup attained is less than 30%. For providing broad insights, we classify unaccelerated regions through manual inspection of their associated source code into five categories, which we describe below.

- **Memory Latency Bound** These regions exhibit indirect memory access, like tree traversal or pointer chasing. Because the computation for these regions is generally very lightweight, accelerators should likely remain as simple as possible, while achieving the same performance as the baseline core.
- **Loop Parallel + Irregular Memory** These regions exhibit significant loop parallelism, but are ineffective for architectures which exploit vectorization, because memory access must be performed in a scalar fashion. The best way to target these architectures may be through scatter gather techniques.

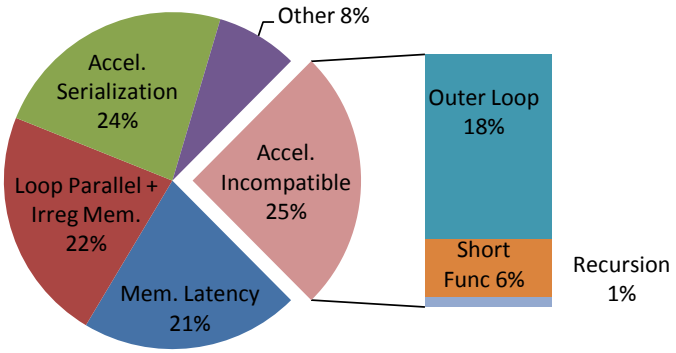


Figure 7: Breakdown of Unnacceleratable Regions (those which have $< 30\%$ performance improvement)

- **Excess Accelerator Serialization** These regions are characterized by excess control flow, making them ineffective for vectorization, or by significant memory access, which tends to serialize the structured hardware accelerators which we considered.
- **Accelerator Incompatible** These regions are those which are incompatible with the set of accelerators we have, because the target certain program structures. These include nested loops, short-but-frequent functions, and recursive functions.
- **Other** This category is a catchall for regions which have more complex interactions which could not be precisely determined, or if the corresponding source was unavailable.

Analysis Our first result is that 50% of the program's regions, in terms of their contribution to the overall execution time, cannot be accelerated by given accelerators. This suggests that there may be ample opportunity for new accelerator designs. To go more in depth, Figure 7 shows the breakdown of various regions types with the above categorizations, for which there were 73 regions.

Roughly equal portions of the programs execution fall into the above four categories. First, the fact that many of the unacceleratable regions are attributable to high memory latency and irregular memory in loop parallel regions, means that extremely simple accelerators and scatter/gather extensions to SIMD are indeed of significant value. Also, many of the regions fall into the excess accelerator serialization category, meaning that too much control or memory access is preventing them from reaching better performance than the OOO core. Most interestingly, however, is the fact that many regions are not able to be accelerated because of the program structures which are used, and specifically outer loops seem to pose the most impactful problem.

Program Structure Breakdown To shed more light on the various program structures used in programs, Figure 8 categorizes every dynamic instruction into one of several categories: *hot traces* which are single paths through inner loops, *flat inner loops* which are inner loops with no function call boundaries, *inner*

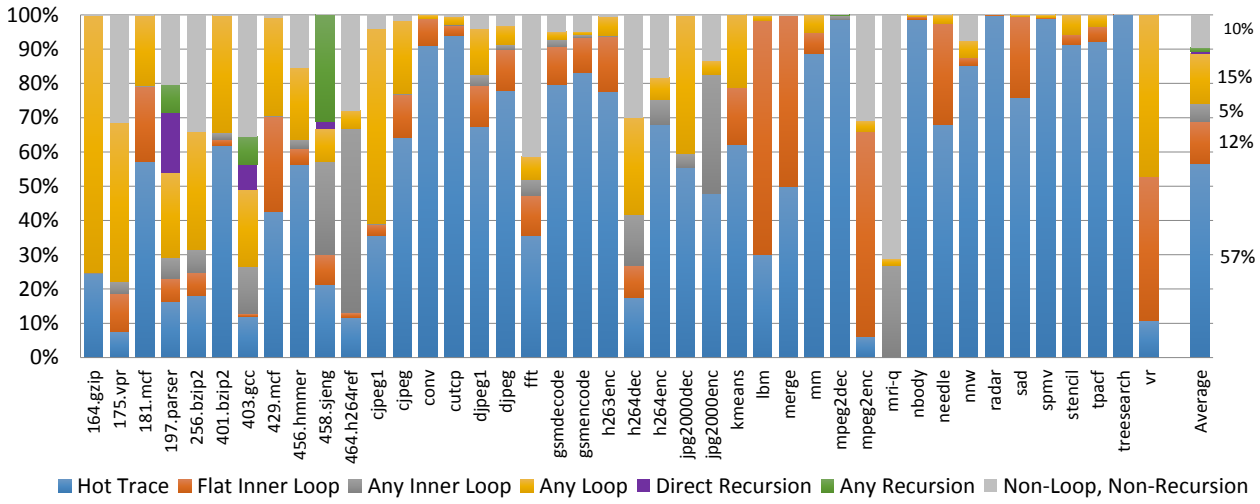


Figure 8: Program structure break down in percentage of dynamic instructions.

loops, *any loop* which includes all forms of nested natural loops, *direct recursion* from one function to itself, and *any recursion*. The remainder of dynamic instructions are those which occur in functions while not in a loop or recursion, typically because of many repeated calls.

As Figure 8 shows, the program structure types vary significantly by program, but on average hot traces are very common, about 57% of the dynamic execution. Most interesting is that, on average, more than 20% of the dynamic instructions happen inside more general loops structures. On current accelerators, these instructions cannot be efficiently executed.

3.3.2 High Level Design: Nested Loop Accelerator (NLA)

To demonstrate the ability of the TDG to perform the high-level design tradeoffs for new architectures, we pick a specific "domain" and architectural style. Leveraging the insights that nested-loops are important across many benchmarks, are exploitable and are under-targeted in existing accelerator designs, we will focus on them here. We also need to choose some basic aspects of the architecture to guide the exploration. Using the insight from the limit study in section 2, reconfigurable hardware accelerators can provide significant energy-efficiency benefit. We choose to explore clustered-instruction execution, like that of BERET rather than the more grid-style fabric of DySER, as the latency of computation will be more important for non-DLP codes which we target here.

We briefly elaborate on the hypothetical architecture and compiler design, and then describe how the new accelerator's design space can be modeled and explored using the TDG.

Hardware Overview Fundamentally speaking, NLA is an in-core accelerator, based on clustered functional unit computation, targeting general purpose execution of nested-loops. It can take full control from

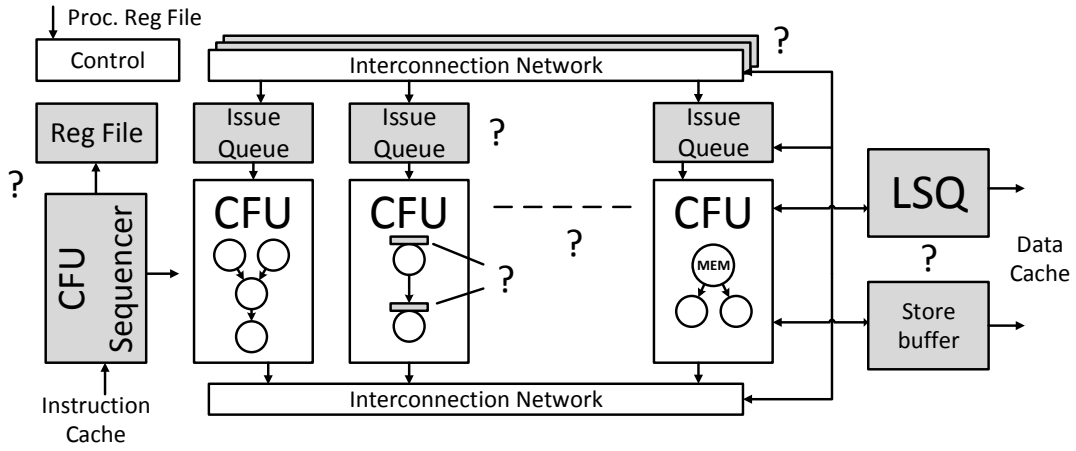


Figure 9: Hardware architecture for NLA accelerator. "?" show potential high-level design decisions.

the main processor when the region is entered, and returns control when the region is complete, transferring state to and from the processor's register file on the transfer of execution. It achieves energy efficiency through the compound functional units which can eschew register file access, and by not requiring frontend pipeline stages (fetch/decode/rename/dispatch) on each instruction. Figure 9 shows a hypothetical block diagram for NLA with many design aspects left undecided. Some examples are, "should NLA communicate through a centralized register file, or should state be distributed?", or "should NLA allow multiple memory requests through an LSQ-like structure, or should it serialize and just use a simple store buffer." A later section organizes these into specific design points.

Gathering NLA Meta-information To use our framework to evaluate an architecture, we need to consider a hypothetical compiler design, which we refer to as "gathering meta-information." The most important concern for a clustered-FU accelerator is how to partition the instructions onto the various compound FUs. This somewhat resembles the spatial architecture scheduling problem of [46], and we use parts of the solution for our initial NLA scheduler

We briefly formally describe the NLA scheduling problem, and present a simple integer linear program (ILP) to solve it. Consider a computation graph made of vertices $v \in V$, where $G_{v_1 v_2}$ represents data dependences between vertices v_1 and v_2. Similarly, consider a hardware graph of computational resources $n \in N$, where $H_{n_1 n_2}$ represents the connections of the hardware substrate between n_1 and n_2. The compatibility between hardware resources and computation vertices is given by the set C_{vn}. The goal is to attain a mapping M_{vn} from computation vertices to hardware nodes such that the number of edges in G not mapped to edges in H is minimized, essentially minimizing the interconnection network traffic between compound FUs. We give the ILP to solve this problem in Table 2. Proposed work describes making this solution even more general and aggressive. In general, integer linear programming is a good fit for an optimistic compiler,

as it can guarantee important solution properties.

ILP Equation	Explanation
$\forall_{v \in V} \sum_{n \in C_{vn}} M_{vn} = 1$ $\forall_{v \in V} \sum_{n \notin C_{vn}} M_{vn} = 0$	These equations enforce that all computational vertices are mapped to exactly one compatible node.
$\underset{\substack{v_1 v_2 \in G \\ n \in N}}{\forall} B_{v_1 v_2} \geq M_{v_2 n_2} - \sum_{\substack{n_1 \in C_{v_1 n_1} \\ n_1 n_2 \in H}} M_{v_1 n_1}$	This constraint computes a set of binary variables $B_{v_1 v_2}$ which describe whether two computation vertices map to the same node. If B is false, then there is no "boundary" between the nodes, and they execute in the same instance of a CFU.
$\underset{v_1, v_2, v_3 \in P_{v_1 v_2 v_3}}{\forall} (1 - B_{v_1 v_2}) + (1 - B_{v_2 v_3}) - 1 \leq (1 - B_{v_1 v_3})$	This constraint enforces boundary transitivity: if there is no boundary between v_1 and v_2, and there is no boundary between v_2 and v_3, then there can't be a boundary between v_1 and v_3. It uses the set P, which contains members of V which are possible to map to each-other, which is computed offline using a simple graph traversal.
$\underset{v_1, v_2, n \in C_{v_1 n} \cap C_{v_2 n} \cap P_{v_1 v_2}}{\forall} M_{v_1 n} + M_{v_2 n} \leq B v_1 v_2 + 1$	This constraint makes sure that two nodes which could possibly map to each other, are only allowed to map to the same hardware node if they are on the same CFU.
$R = \sum_{v_1, v_2 \in G} B v_1 v_2$	This equation defines R, the number of either register file reads, or interconnection network accesses. R is the objective of minimization.

Table 2: Integer linear programming model for NLA Scheduling.

3.3.3 Design Space Exploration

Table 3 below describes four NLA design points, from most simplistic and presumably energy-efficient, to most complex and high-performance. The basic hardware required is briefly described in the first column, and the second column describes the TDG transformations required to model these mechanisms, or hypothetical features. Note that there is a close correspondence between hardware mechanism and dependence graph transform, enabling very fast exploration of architectural designs. This setup is related to and inspired from the limit relaxations in the previous section. The difference here is that, while we only show four designs for brevity, in general, the cross product of possible design decision/graph transformations are useful and interesting for the designer, and will be evaluated.

In our design space exploration, in addition to evaluating NLA's design points, we also evaluate existing accelerators. In Figure 10 we show preliminary results for six of the benchmarks from PARBOIL and TPT benchmark suites, using the four design points described earlier. The first two benchmarks, cutcp and mm, exhibit accelerator benefit overlap with all NLA designs, because either SIMD or DySER is more effective, even for the most aggressive NLA designs. For the second two benchmarks, nnw and merge, for all but the

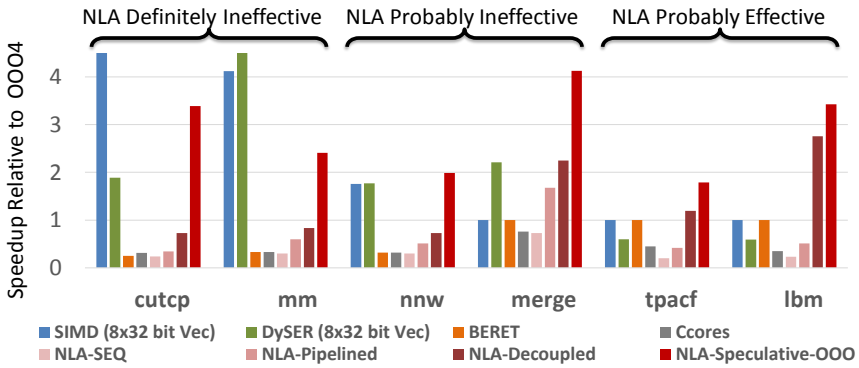


Figure 10: Preliminary Results for NLA-Design Space Exploration

most aggressive speculative-OOO version, NLA's benefit is masked by other accelerators. For these four benchmarks, we no longer need to consider NLA as a potential accelerator design. Looking at the last two benchmarks, tpacf and lbm, the practical NLA-decoupled design actually performs well compared to the other accelerators and the general purpose processor. Finally, we can also observe than the sequential and pipelined designs are ineffective for competing with an OOO core.

Overall these experiments have shown how we can eliminate potential designs and many benchmarks from consideration, all with the use of extremely high level models and low design-effort. Also, though we have only shown four design points for the sake of brevity, we can make and evaluate design choices independently, so we can explore many more designs very rapidly.

Design Point	Hardware Implications	TDG Transformations
Serialized	• Serialized hardware implies simple design, no arbitration need. • Regfile stores results bet wen each invocation. • Forwarding network for passing values used by subsequent instructions.	• Edges inserted between the end and beginning of subsequent CFU nodes. • Dependence inserted between memory address calculations in program order.
Pipelined	• On each cycle, Sequence Unit issues the next compound instruction if its operands are ready and CFU is not busy. • Writeback bus must be arbitrated so only one CFU can write at a time.	• Edges are inserted between the CFU invocations. • Control operations serialize the next CFU invocation. • CFUs are treated like resources, and dynamic edges are added between nodes requesting contended CFUs.
Decoupled	• Dataflow architecture with distributed issue queues across CFUs. Issue queues fire when operands are ready and CFU is free. • Software disambiguates the memory addresses by static load/store where possible, unknown accesses are serialized. • Interconnection network is multi-issue point to point. • CFUs are pipelined for additional parallelism.	• Only CFU serialization edge is for enforcing control dependence. • Edges inserted only between memory-operations which may alias through software analysis. • Interconnection bus treated as resource, dynamic edges inserted during contention.
Spec. OOO	• Full speculative support allowing execution past control nodes. • Mispeculated instructions squashed. (hypothetical mechanism) • Memory dependence prediction. (hypothetical mechanism)	• No CFU serialization edges. • Squash penalty enforced by multi-cycle dependence between complete of mis-speculated control and subsequent instruction. • Squash penalty inserted for mispeculated memory dep prediction.

Table 3: Description of NLA Design Points and their Models

3.4 Proposed Work

The proposed work falls into several categories. For the NLA accelerator, there is essentially work left in the third and fourth design phases. We also will explore two more-speculative ideas.

Exploration Phase The exploration space should be increased in terms of architectural features. Also, running a larger, more representative set of benchmarks will be necessary. On the compiler side, it can be extended in a variety of potentially useful ways. The scheduling problem, as described earlier, can only minimize the network activity, and does not attempt to optimize latency or throughput by considering the utilization/contention of resources. Attempts to model these features have so far produced integer linear programs which are too slow. The proposed work will explore whether ILP for more aggressive scheduling is possible.

Refinement Phase To complete step 4 of the accelerator design strategy outlined earlier, we will iteratively design the "low-level" hardware features. For example, protocols for handling full buffers in the dataflow network, which can prevent deadlock and allow forward progress, are necessary. Another example would be the mechanism which forwards the memory access token between memory which may alias. Once mechanisms are decided upon, then dependence nodes and edges for these features can be added, and more realistic models can be attained.

Automating Analysis One of the most pressing issues is in how to give more insight to the designer to inspire interesting and useful accelerator designs, as well as insight to why certain designs are better than others. One possibility is to characterize "unexplored" or "high-benefit" applications automatically into groups based on certain observed program characteristics, in an attempt to make analysis easier. Clustering algorithms may be able to be employed for this purpose. Another useful feature would be in understanding fundamentally why certain designs or design variations perform differently. One possibility is to use the critical path through the graph to see what dependency types are most prevalent in a certain execution. By associating dependency types with architectural features, we should be able to discern what is the prevailing cause for various speedups and slowdowns.

Further Accelerator Designs Finally, in Section 3.3.1 we discovered several categories of program regions which go unaccelerated in important applications. Part of this dissertation will explore whether there are further opportunities for new acceleration paradigms.

Also, it is important for evaluating the design process to isolate the benefits of the particular design we target with the benefits design process itself. One way to do this would be to re-design an accelerator which

we have developed using standard practices, then qualitatively compare the difficulties of each approach, and quantitatively compare the design quality. One possibility is to use our research group's "MAD" accelerator.

3.5 Related Work

ADLs for Compiler/Architecture/Simulator Generation One body of particularly related work uses architecture description languages (ADLs) as a specification tool for rapid architectural exploration and evaluation. UPFAST is a tool for generating a simulator and assembler/disassembler, mainly used for extensions to MIPS-based architectures [49]. BUILDABONG is a tool for architecture compiler co-exploration, requiring architectural specification with an abstract state machine, and can generate HDL models, simulators and compilers [22]. Productized approaches include the Tensilica Instruction Extension language [26], which allows for the manual creation of specific instructions which can be automatically incorporated into existing architectures and compilers. For a comprehensive survey on automatic instruction-set extensions, see Galuzzi et al. [24]. Another example is the Language for Instruction Set Architectures (LISA) [33], which can be used to generate compilers, assemblers and functional simulators, as well as hardware implementations.

The Liberty Simulation Environment [55] is a somewhat unique tool which can be used to create simulation models which are more accurate, and as the authors claim, more intuitive and maintainable than those created using sequential languages. A language called the Liberty Structural Specification [54] is used to construct models of hardware components which are connected through explicitly defined interfaces, rather than the implicit interfaces of the function-abstractions which sequential languages provide.

While the above approaches can aid the design, exploration and even generation of architectures, simulators and compilers, they are only applicable for certain types of instruction-by-instruction, Von Neumann style architectures. For accelerator design, we need something more general and powerful.

High Level Synthesis High-Level Synthesis (HLS) attacks design inefficiency from the bottom up: its goal is to make the development of hardware designs easier by using languages which are higher level than either Verilog or VHDL. Typically, these tools take hardware designs written in high level languages like C,C++ or SystemC, and produce optimized, synthesizable RTL [40]. Because sequential languages aren't necessarily effective for expressing hardware designs, another HLS tool, Bluespec [44], uses a custom language based on Haskell and SystemVerilog.

We see our proposed approach as being largely compatible with HLS tools, which can take over to explore more fine-grained, implementation-specific tradeoffs after the broad design-space exploration that our approach allows.

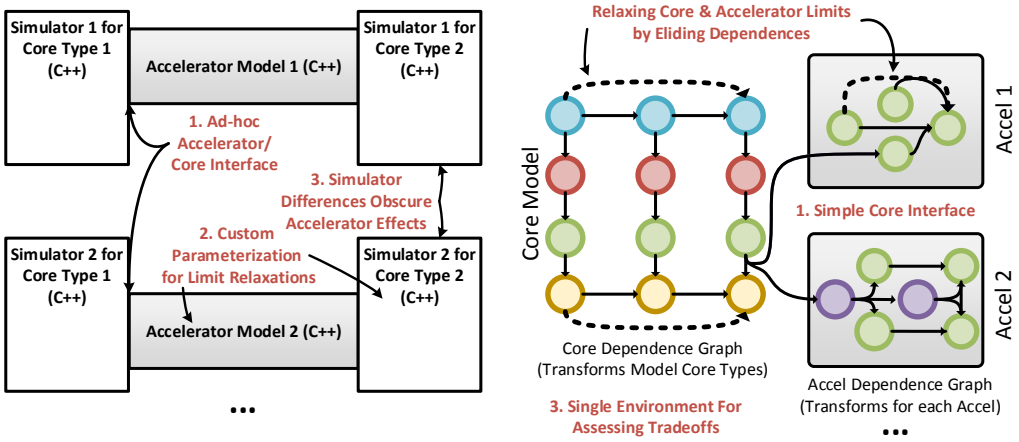


Figure 11: Current versus proposed approach for finding Limits/Opportunities of Acceleration.

Accelerator Design Processes In terms of accelerator design strategies, we are certainly not the first to recognize that designing for multiple accelerator systems are important. The most closely related strategy is the 10×10 framework [8]. Their essential strategy is to describe clusters of applications which exhibit certain behaviors, and develop in-core "micro-engines" (accelerators) which target each domain. The key difference between our approaches is that they separate and delineate benchmarks into categories *a priori*, while our approach can find patterns across predefined benchmark "categories," and therefore can find a more optimal, and likely more lean set of accelerators.

Architecture In terms of the hardware design, NLA borrows many concepts from the BERET architecture [30]. However, it's prospective design and intended purpose differ in several key aspects. First, it targets nested loops, which are much more general than the hot-traces which BERET targets, and allows for longer accelerator-only periods where the core can enter into lower power states. Also, it has the potential to deserialize aspects of the execution which would can lead to performance levels useful in OOO processors.

4 The Limits and Opportunities of Acceleration

4.1 Motivation

If the future of performance and energy efficiency entails specialization through accelerators, then it becomes vital to understand both what type of accelerators should be developed and what challenges are the most important to tackle. Essentially, architects need to know where to look, and how to spend their time. As Figure 11 outlines, this is extremely difficult given the current state of accelerator research, where different accelerators are implemented in different simulators, requiring vast effort in implementation as well as making

it difficult to compare results. The transformable dependence graph (TDG) is in a unique position to help on this front; it is both abstract and can be easily provisioned to model new design choices and forms of acceleration, and at the same time can accurately capture low level interactions with the application and with the general purpose processor. The abstract nature of the TDG helps in an additional way: we don't need to restrict ourselves to designs which can be practically built today, we can instead look at the vast array of architectures which break dependences which we could forseeably invent techniques for in the future.

What we propose in this direction of research is to describe a set of "limitations" which are the primary factors which prevent accelerators from realizing their full potential performance or energy-efficiency benefits. These limitations fall under broad categories like reducing memory serialization or communication with the core, but will have different hardware implications and expected benefit for different accelerators. Implementing TDG models for the different limitations, for the most part, entails omitting certain nodes and dependence edges. We are essentially turning off parts of the models which are already there, where the upper bound model for an accelerator leaves only the computation nodes, and the data and memory dependences.

We also clarify that the our goal is to explore the limitations and opportunities of acceleration which can be applied *automatically*, without the aid of a programmer to revise the application's algorithm. That is, what are the fundamental tradeoffs, limitations, and potential benefits of accelerators attainable through automated processes alone.

4.2 Research Approach

After performing detailed analysis across many benchmarks and potential limitations, we present 9 limitations, both practical and fundamental, explained next, and outlined in Table 4. The ordering of the transformations will certainly affect the perceived benefit of each component. We have attempted to order them in increasing difficulty to address, which makes sense because it's rational for designers to address lower hanging fruit first.

Practical Limitations

- **Vector-Fetch Width** Vector fetch width is the amount of data which is fetched to or from memory with a single instruction.
- **Accelerator Size** In this limitation we vary the most fundamental aspect of the architecture which affects it's total area. Note that these are not normalized to eachother as they are meant to be a sensitivity parameter. For Conservation Cores we the number of static operations mapped to hardware.

Limit	Description	Parameters
	Practical Limitations	
Vector-Fetch Width	Amount of data fetched to/from memory with one instruction	4-words, 8-words, 16-words
Accel. Size	Area of Accelerator (note, these are not normalized to eachother)	Beret SEB size: 5,7,9,12; SIMD Lanes: 4,8,16 C-Cores K-Ops: 0.15,0.5,1,8; DySER FUs: 16,25,32,64
Mem. Serialization	Maximum parallel memory ops.	Beret/CCores: 1,2,4,8 Mem Ops per memory region.
Core Comm.	Overhead of data-transfer.	DySER: 3, 1 cycle instructions & Free BERET/C-Cores: 1,2,4 Ops/Cycle
Accel configuration.	Overhead of accelerator configuration time	DySER: 64 cycles & Free BERET: 1 cycle per SEB & free
	Fundamental Limitations	
Block Dataflow	Acceleration unit blocks are de-serialized.	On/Off
Full Dataflow	Full dataflow for entire acceleratable region.	On/Off
Execute Specialization	Reduces long latency FU access, approximating special function unit integration.	4,1 cycle max latency
Memory Specialization	Allows the working set to fit into a particular level of cache.	Maximum latency mem access: L2, L1 cache latency

Table 4: Summary of Limits Considered

For DySER, it's size is proportional to the size of the functional unit (FU) grid. For BERET, the SEB sizes, in terms of FU count, are considered. For SIMD the number of compute lanes is the biggest factor.

- **Memory Serialization** In this limit, we consider the number of allowed parallel memory operations. This only pertains to accelerators which do not borrow the main processor for memory, namely C-Cores and Beret.

- **Core Communication** Here we consider the overhead in communicating values between the core and the processor. For BERET and C-Cores the data transfer model is calculated in the number of words/cycle. For DySER, this is the execution latency of the send/recv instruction.

- **Configuration Overhead** This limit relaxation reduces the overhead time of setting up the accelerator upon transferring control to the main core.

Fundamental Limitations

- **Block Dataflow** Each accelerator executes in "blocks" of computation. For Conservation Cores this is the basic block, for BERET the SEB, and for DySER this is the computation subregion. This relaxation allows multiple blocks to be executed in parallel, if it isn't already allowed for the architecture. Specifically, any ready blocks may fire once the current basic block has completed. In practice, while this requires potentially more complex parallel hardware, it does not impose additional control speculation on the program.

Suite	Benchmarks
TPT	conv, merge, nbody, radar, treesearch, vr
Parboil	cutcp, fft, kmeans, lbm, mm, needle, nnw, spmv, stencil, tpacf
Mediabench	cjpeg, djpeg, gsmdecode, gsmencode cjpeg2, djpeg2, h263enc, h264dec, jpg2000dec, jpg2000enc, mpeg2dec, mpeg2enc
SPEC INT	164.gzip, 181.mcf, 175.vpr, 197.parser, 256.bzip2 429.mcf, 403.gcc, 458.sjeng, 473.astar, 456.hmmer
NPU Suite	fft, inversek2j, jmeint, jpeg, kmeans, sobel

Table 5: Benchmarks

- **Full Dataflow** Full dataflow means that all operations which are designated to be run on the accelerator are have no other resource or scheduling restrictions on them. Note that this assumes enough hardware for all simultaneous basic blocks to be executing in parallel, as well as facilities to predict branches. For BERET this means speculating whether loop iterations will be taken, for C-Cores this implies branch prediction. We assume the C-Cores model has the same type of branch predictor as the main core, and serialize on a misprediction.

- **Execute Specialization** This relaxation enables long latency functional unit access to be reduced, which approximates the benefit of integrating special function unit access for relevant computation. We use values of 4 and 1 cycles max FU latency.

- **Memory Specialization** This allows the working set of the program to "fit" into either L2 or L1 caches. We implement this by clamping latencies to some max value.

4.3 Preliminary Results

We first briefly describe some of the basic methodology and setup, then discuss preliminary results.

Machine Parameters We list here some common machine parameters. We use a 2way set-associative 32KiB I$ and 64KiB L1D$, with a 4 cycle latency, and a 2MB 8-way assoc, L2$, with a 22 cycle hit latency. The core has a 192 entry ROB, 64 entry IW, 32 entry LSQ, and 2 load/store ports. The four-issue OOO core has 3 ALUs, 2FPs, and 1 Mul/Div unit, which are scaled to various issue widths. For power and energy, we use the $22nm$ setting in McPAT.

Benchmarks We selected benchmarks from several suites, including throughput kernels from [27]. MediaBench [36], Parboil [1], and SPEC INT, which are listed in Table 5. This set of benchmarks covers applications from many domains, and their diversity serves to demonstrate application-accelerator interaction.

Cross Product Results Figure 12 shows summary of the limit study between the cross product of benchmark, general purpose core and accelerator, using the metric of energy-delay. On the y-axis is the

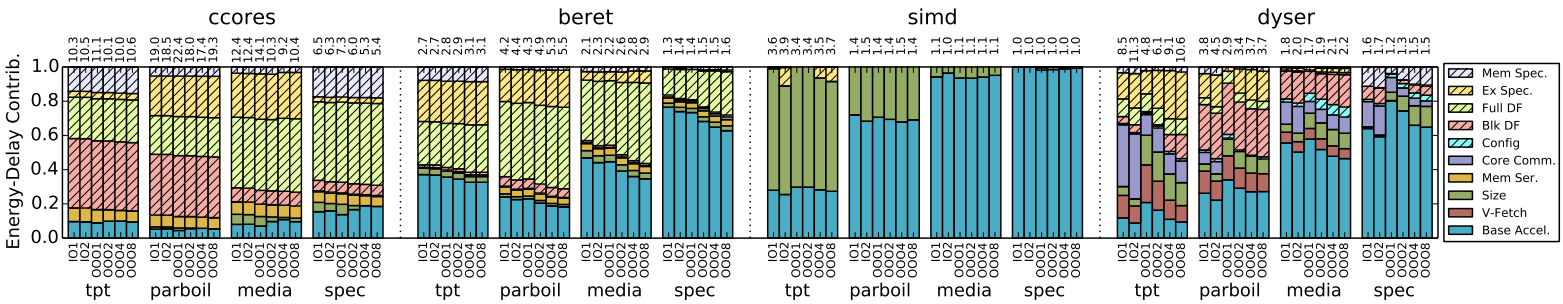
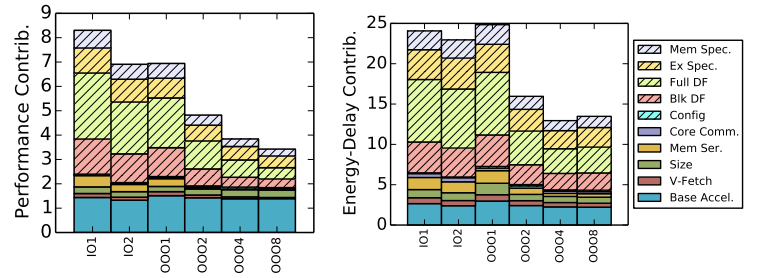
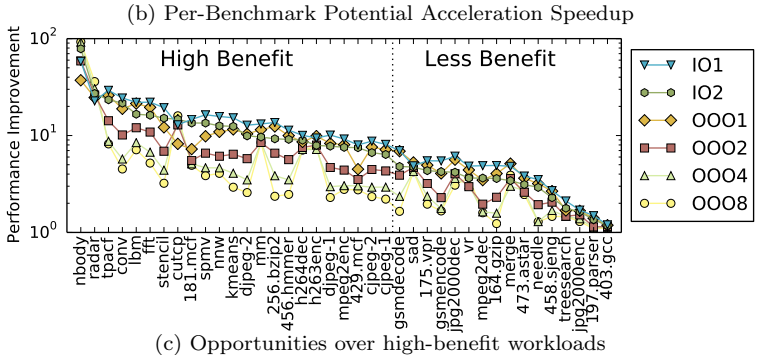


Figure 12: Impact of Limit-Relaxations for all Cores/Accelerators/Benchmark suites, in terms of normalized energy delay. Numbers above bars indicate maximum energy-delay achievable.

(a) Opportunities over all workloads



(b) Per-Benchmark Potential Acceleration Speedup



(c) Opportunities over high-benefit workloads

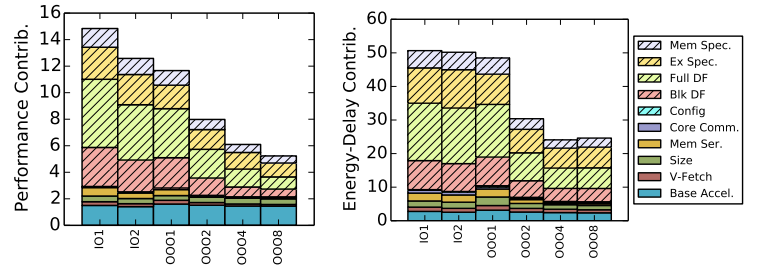


Figure 13: Opportunities across all Benchmarks

percentage of total improvement which each limit relaxation provides, and the number above each bar is the total improvement.

The first observation is that relaxations have different effects on different benchmark suites. For conservation cores on data-parallel benchmarks like TPT and PARBOIL, enabling block dataflow is enough to see 3× improvements in energy-delay. The same architecture with SPEC INT, which has much more frequent control flow, needs full-dataflow to be effective. Also, while one would expect the benefit of relaxations to be

independent of the GPP core type, this is clearly not always the case. Considering the DySER accelerator, we can see that the core communication bottleneck can provide huge improvements on certain benchmarks, especially on the inorder-core types.

Benefits of Multi-acceleration By combining multiple accelerators together, it's possible to see significantly more benefit than from just a single design alone. The true potential for acceleration lies in the ability to combine techniques, achieving the benefits of heterogeneity on a much more fundamental basis than just microarchtiectural changes. To see the potential benefits of such an approach, Figure 13(a) shows the geometric mean improvement when considering the best possible acceleration for each benchmark, relative to its baseline general purpose core for each relaxation we consider. Figure 13(b) shows the potential performance improvement for each core type on each benchmark, considering the best accelerator for each. If we consider benchmarks with over 5× speedup on inorder cores as "high-benefit" benchmarks, then Figure 13(c) shows the improvement of acceleration on these.

Here we can see that reducing serialization, both inside and across accelerator unit blocks, is hugely important and can lead to significant improvements in both energy and energy delay. To summarize the results, across all benchmarks, the multi-accelerator potential is up to 3.5× and 8× performance for OOO and inorder cores, and up to 13× and 25× respectively for Energy-Delay. For high benefit workloads, the potential is 5× and 14× performance on OOO and inorder cores, and up to 30× and 50× for Energy-Delay.

4.4 Proposed Work

Additional Experiments & Analysis While we have merely scratched the surface in this proposal, our ultimate aim is to comprehensively address the following five questions about the nature of acceleration. 1. What are accelerator's fundamental limitations? 2. What are the potential benefits of multi-accelerator systems? 3. By how much does the baseline general purpose core matter? 4. Is the metric important in choosing an accelerator? 5. What accelerators are effective in which domains? Using the transformable dependence graph is an appropriate approach, because it does direct modeling of the execution of an accelerator on a benchmark as well as modeling the GPP, so in can capture application and core interactions, and also is abstract enough to be flexible in modeling. The experiments for questions 4 & 5 are mostly explained. For questions 1-3, we will pick a specific design point for each accelerator, and then vary the applications and baseline cores, and analyze the behavior of the models to find patterns.

Formalizing Transformations We also propose to more formally define the semantics of the graph transformations for modeling accelerators and performing limit studies. This work so far essentially describes how

the graph is changed by a transformation, and would be significantly improved by a more formal algorithmic description of the transformation procedure. Even better would be to describe the transformations in such a way that they can be guaranteed to be correct, even when composed in some ordering. While it's not certain whether a framework such as this exists, its possibility merits exploration.

4.5 Related Work

Modeling Many recent papers have presented a case for specialization and heterogeneity using analytical models [10, 62, 32], high-level technology trend projections [6, 35], or by modeling general purpose processors [16]. Models for heterogeneous execution include [48, 47, 60, 34, 53, 41, 59, 42], which are related to analytical models for general purpose processors [18, 15]. These are specific to one accelerator or general-purpose core performance and do not allow projection of other metrics like power and energy or limit studies. Cascaval et al. introduce a taxonomy and outline some opportunities of acceleration [7].

Quantitative/Limit studies Azizi et al. present a cost analysis framework of optimal general purpose processors from an energy and performance perspective [3]. In the previous decade, Agarwal et al. presented a study on fundamental limits of OOO microprocessors [2], and preceding that, the limits of ILP [58]. Our work, is a reincarnation of Wall's ILP study for today's accelerators/heterogeneity driven design paradigm.

5 Modeling Acceleration with Dependence-Graph Transformations

This section describes our methodology for modeling acceleration, which leverages the proposed transformable dependence graph (TDG). We first give insight into the potential of dependence-based models, and discuss background material. We then describe how the TDG works, along with specific transformations and validation for five architectures, and conclude by discussing the scope of the TDG.

5.1 Why Dependence-Graphs?

The Nature of Acceleration We first define an accelerator as: *hardware which executes work in lieu of the general purpose processor to specialize elements of its execution.* This means that the benefits and effectiveness of acceleration depends what and how the general purpose processor gets specialized (**accelerator-specialization**), the application for presenting certain characteristics which can be specialized (**accelerator-application interaction**), and also in how the accelerator communicates with the core for coordination (**accelerator-core interaction**). Any effective model of acceleration must capture these three aspects.

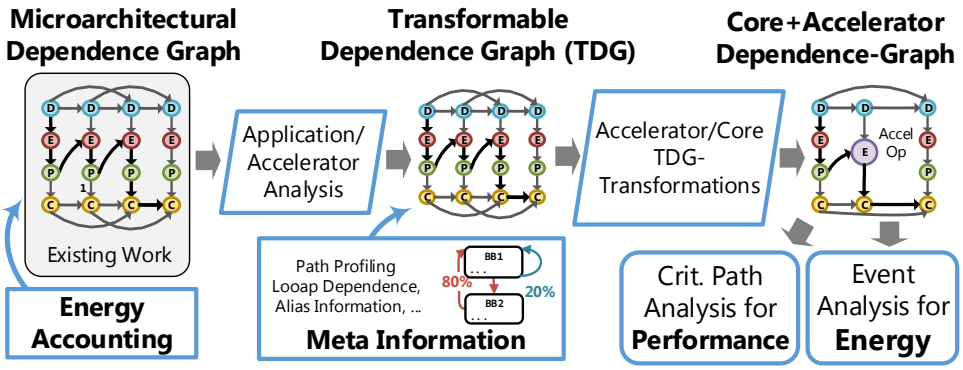


Figure 14: Applying the Transformable Dependence Graph

Dependence-Graph Models Dependence-graph models are representations of the trace of execution for an architecture running a specific application, which are composed micro-architectural events as nodes, and dependences between events as edges [21]. Edges have a weights that represent the latency between events, and represent phenomenon like the issue width, ROB size, and data dependences.

The dependence graph is first of all a great candidate for accelerator modeling because the dependence edges are exactly the aspects of general purpose cores which can be specialized, and modeling this specialization is as simple as eliding these edges (capturing accelerator-specialization). Next, the dependence graph itself is a representation of the program, containing all the data and memory dependence information which characterizes it. For this reason, analyzing the program and the way the acceleration is affected by the program is natural (capturing accelerator-application interaction). Finally, since the graph represents the execution at a sub-instruction granularity, the communication between the accelerator and core can be modeled with simple additional dependence edges (capturing accelerator-core interaction).

Therefore, the key to using dependence-graph models for acceleration lies in how to transform the existing dependence graph representing general-purpose architectures to model the unique and varying phenomenon experienced by accelerators.

5.2 Overview

In this work, we propose a new abstraction, based on microarchitectural dependence-graphs, for studying the nature of acceleration called the *transformable dependence-graph(TDG)*. This abstraction augments the original dependence-graph model with the ability to apply transformations which model the dependencies inherent in various accelerators. Figure 14 shows how our approach employs the TDG for modeling acceleration. First, we take an existing microarchitectural dependence-graph, and augment it with annotations concerning the type of operation and accesses it performs, which can be accumulated and used by existing power models to calculate energy. Next, an analysis pass determines specific application properties, which

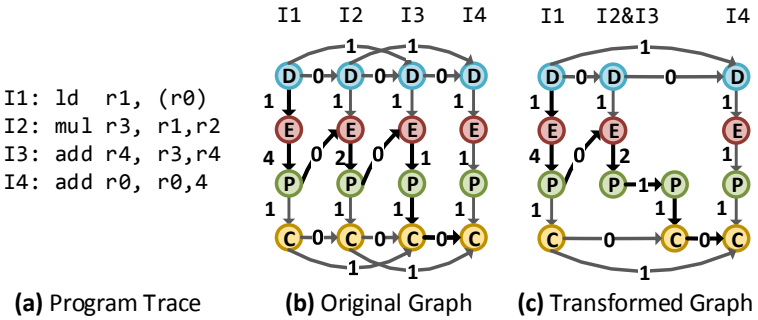


Figure 15: Using the Transformable Dependence Graphs for modeling Custom Instructions. Edge descriptors in (a) and (b) indicate latency. Legend – D: Dispatch, E: Execute, P: Complete, C: Commit

we call *meta-information*, to determine when it is legal and beneficial to perform acceleration. We then transform the graph according to rules which model the accelerator's behavior, modifying or inserting event nodes and dependency edges. Using this transformed graph, we determine the performance and the energy of the overall execution.

5.3 The Transformable Dependence Graph(TDG)

Microarchitectural Dependence-Graphs Background Microarchitectural dependence-graphs [21] represent the dynamic execution of processor pipelines abstractly. Nodes in the graph represent microarchitectural events, and edges in the graph represent static and dynamic dependencies between events. To give an example, Figure 15(a) shows a series of instructions, and Figure 15(b) shows a simplified dependence graph representation of the execution for a two-wide out-of-order processor.

Nodes here represent different stages of the instruction's execution, like "dispatch," "execute," "complete" and "commit." Edges represent dependences which enforce constraints imposed by the architecture. To explain the diagram, edges between subsequent dispatch and commit nodes have a zero cycle latency, and represent that both dispatch and commit must occur in order ($F_{i-1} \xrightarrow{0} F_i$, $C_{i-1} \xrightarrow{0} C_i$). Edges between alternate dispatch and commit nodes enforce the issue width of the processor ($F_{i-2} \xrightarrow{1} F_i$, $C_{i-2} \xrightarrow{1} C_i$). The edge from execute to complete enforces the latency of the operation or load ($E_i \rightarrow C_i$). Finally, data dependencies are edges between instructions, from complete to execute ($P_i \xrightarrow{0} E_j$). By traversing edges through the graph, we compute the critical path and thus the performance.

Using the Dependence-Graph to Calculate Energy One of our contributions is to show how the microarchitectural dependence graph can be used to calculate energy. We take a quite straight forward approach in associating appropriate energy events with each node. For instance, the dispatch node accesses the ROB, the execute node accesses specific functional units (FUs), and so on. As we traverse the dependence-graph, we calculate the total event counts of each type. Much like a simulator, we can feed these event counts

into established tools for energy/power estimates.

We also note that since the graph does not contain mis-speculated instructions, we also must calculate the estimated number of additional events that would have occurred. We do this by calculating the ratio of speculated to misspeculated instructions, and multiplying the overall counts by this number.

Enabling Transformations to the Dependence-Graph The fundamental insight which we employ in this work is that accelerators, which specialize elements of the processor's execution, can be modeled by removing and transforming nodes and edges. Our evaluation shows that using the TDG, we can achieve less than 15% average error for all accelerators against published or simulator data for both performance improvement and energy reduction.

To demonstrate this concept, we can consider adding a custom multiply-accumulate instruction. The TDG can capture this by recognizing this computational pattern in the application, and eliding the dispatch and commit nodes from the redundant instruction with a transformation. Edges connecting to the surrounding instructions are reattached to capture the pipeline constraints, and Figure 15(c) shows the resulting graph. What enabled the transformation for the above example was the *meta-information* that the original two operations fit the correct computational pattern. In general, what makes the dependence graph "transformable" is having the appropriate meta-information about both the semantics of operations and properties of the dynamic execution. Which meta-information is required depends on the accelerator, and we discuss the next section.

5.4 Accelerator TDG-Transformations

Here we describe how we modeled five specific accelerators through applying TDG transformations. These were chosen because they require different program characteristics to be effective, which helps show how the TDG can capture application-accelerator interactions. Brief descriptions follow.

Conservation Cores, is a structured hardware accelerator, which automatically generates hardware implementations of application code. It is meant to be an offload engine for energy efficiency.

The **BERET** accelerator is also a structured-hardware accelerator, essentially a trace processor for loops, where only the most frequently executed control path is executed in the accelerator, using efficient compound functional units called SEBs.

SIMD acceleration relies heavily on exploiting independent loop iterations for parallel execution, and is a well known acceleration strategy.

DySER is a CGRA, operating in an access-execute fashion with the processor, accelerating the computational portions of the code. It is integrated with a vector interface to the host.

NPU is a domain specific accelerator, meant to accelerate approximate computations with neural networks(NNs). It uses custom instructions for communication, and an access-execute model.

Table 6, describes the meta-information required and the TDG transformations for each accelerator. Figure 16 parallels this description, showing an example for each accelerator. The original code has inner-control flow inside a loop, and also independent iterations. On the right of the figure is the original dependence-graph of four iterations of the dynamic execution. Each panel shows accelerator-specific transformations.

	Gathering Meta-Information	TDG Transformations
Common	• **CFG&DFG** Static CFG and dataflow graphs are built using the dynamic trace. The CFG is constructed by tracking BB boundaries, including branch targets. • **Loops** Used dominator analysis to construct natural loops, and loop nests.	• **Processor Dependence Elision** Most accelerator's offload instructions, so for these instructions, their processor dependence nodes and edges are elided. • **Processor Instruction Elision** Many μops which the processor executes are not required in dataflow based accelerators, like register moves, so they may be elided.
C-Cores	• **Loop/Func Frequency** Loop and function frequency is derived from the dynamic trace of the program. • **Selected Loops/Funcs** Region selection to cover the maximum dynamic execution with fixed hardware budget. Our heuristic walks the call tree, picking the most profitable loops and functions.	• **Basic Block Serialization** As the accelerator is non-speculative, basic blocks are serialized with additional edges between them. • **Memory Serialization** Memory ops serialized in BBs. • **I/O Transfer** An edge is inserted between the Conservation Core and the processor for transferring values, on entry/exit.
BERET	• **Path Profiles** BERET exploits loop path re-use, so we capture this application interaction with path-profiling [4]. • **SEB Partitions** BERET maps hot-path code to limited # of compound FUs(SEBs). Process modeled by fixed size subgraphs, optimized with integer linear programming.	• **Trace Construction** BERET traces are constructed at appropriate loop heads, BBs serialized with an additional node. • **Region Configuration & I/O Transfer** Control transfer node models RF communication and configuration delay. • **Instruction Replay** On BERET misspeculation, the last iteration of the loop is replayed on the host.
SIMD	• **Loop Dependence** We determine loop dependence by analyzing memory access through each loop-nest. • **Alias Info** May/must alias information between memory operations. • **Stride Analysis** To see if memory accesses follow a simple pattern which is vectorizable, we implement a per-operation memory access stride analysis, and determine the pattern.	• **Loop Vectorization** Vector width number of iterations are coalesced. Peeled loops retain normal dependencies. • **μop Splitting** Vector ops split for smaller vector lanes. • **Control Flow Merging** Diverging control paths are merged in the TDG. Static instructions are played reverse post order. • **Mask Instruction Insertion** Mask instructions are inserted along merging control flow paths.
DySER	• **Program Dependence Graph (PDG)** The PDG is constructed by analyzing the CFG for control dependence. • **PDG Slicing** The PDG is sliced according to heuristics, creating a memory sub-region with loads, stores and dependent computation. The remaining computation is marked as the execute subregion. Subregions are only considered when communication costs are low.	• **Loop Vectorization** Similar to the SIMD. • **Access-Execute Decoupling** Communication instructions inserted at the borders of access and execute subgraphs. • **Configuration Insertion** Configuration instructions (default 64), are inserted into the main processor's pipeline when a new region is encountered. Regions are cached in 4-entry buffer until evicted.
NPU	• **Appproximable Regions** Marked by programmer. • **Region inputs/outputs** Inputs and outputs of the candidate regions, both memory and registers, determined using existing meta-information.	• **Region Elision and communication** Instructions from approximable region elided from main processor. Execution is replaced with NPU invocation node, and communication edges added based on region inputs/outputs.
Inorder-Core	*none required*	• **OOO Resource-Edge Removal** OOO resource dependences removed, branch penalty reduced. • **Serializing Execution** Inorder complete with: $P_{i-1} \xrightarrow{0} P_i$. • **Long Latency Operations** Long latency operations stall the next execute node by the length of the pipeline.

Table 6: Description of Meta-Information and Graph Transformations

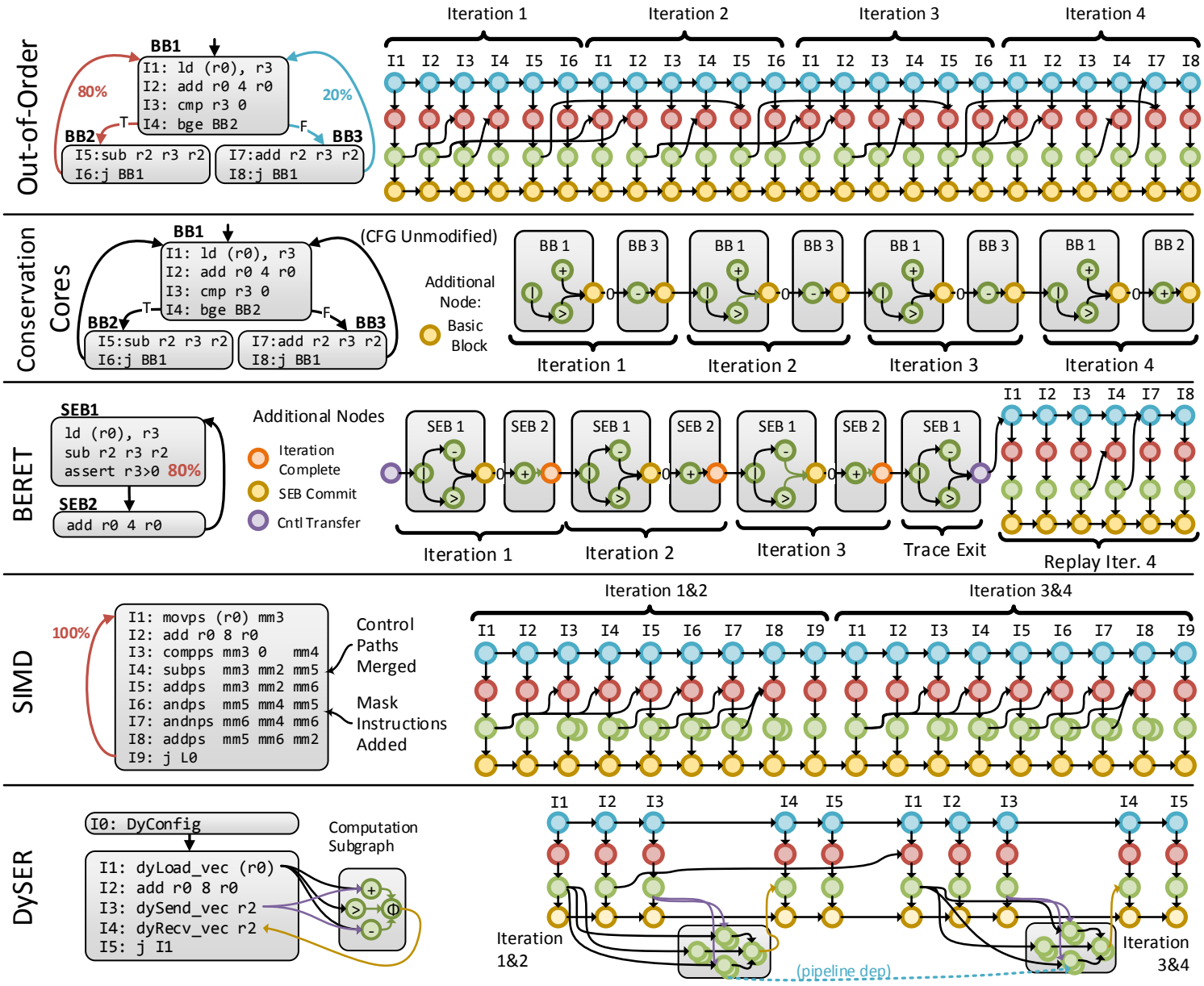


Figure 16: Accelerator Dependence Graph Transformations

5.5 Implementation - Prism

Our implementation of the transformable dependence-graph (TDG) is called *Prism*. We first describe the implementation of Prism, how it integrates with the simulator and energy analysis tool, and then describe the details of the nodes and edges we use to model microarchitectural dependence.

Simulator Instrumentation Prism generates the original dependence graph using gem5 [5]. We gather five pieces of information from the simulator: 1. A description of the static instruction for each dynamic; 2. The dynamic dependencies between instructions (register & memory); 3. The dynamic memory latencies;

4. Branch mispredicts; and 5. Memory addresses. Any other source of this set of information would be sufficient for building the TDG.

Generating Meta-Information and Prism Transformations The second phase of our approach is to generate meta-information by analyzing the trace, acting like an optimistic compiler analysis. A final pass is performed over the TDG, and it is then transformed to model different accelerators, as described by Section 5.4. The cycle counts are computed dynamically by traversing the critical path of the graph.

Energy Models Prism energy models work by first accumulating appropriate energy event counts for both the main processor and also for the accelerator as the TDG is being transformed into core and accelerator components. At the end of the transformation, we have two sets of activity counts. The core activity counts are fed to McPAT [37] for processor power/energy. For the accelerator events, to make the comparison between accelerators as fair as possible, we let McPAT handle common energy components (FUs, reg-files, etc), which it is designed for. We also account for accelerator specific hardware, using per-access energy estimates based on existing publications.

Prism's Dependence Graphs Table 7 shows the nodes that we use to model processor instructions in Prism, and also the intra-instruction edges and their latencies. Highlighted rows are additions to the model of Fields et al. [19]. The biggest difference is the addition of Fetch and Writeback nodes to model front-end pipeline stalls and memory bandwidth respectively.

Of primary note is the additions of the fetch nodes and, for stores, writeback nodes. Fetch nodes were added primarily because it eases the addition of various pipeline stalls. For instance, LSQ full events stall in the dispatch stage, but control misspredictions stall at the fetch stage. Writeback nodes are useful for modeling serialization instructions, and for modeling store bandwidth. In terms of edges between nodes inside instructions, there are other differences to the previous models in the literature. For instance, Prism uses architectural parameters for operation latency and throughput rather than recorded values from the trace, and also we eliminate the latency of cache-line dependent loads, because this recorded latency is simply an artifact of the when the cache-line dependent load happened to get fired. These modifications improve the graphs ability to be accurate when other events are sped up.

Table 8 details the static and dynamic edges that are placed between instructions in prism. Here, static means that the edge is static with respect to the trace of execution. Dynamic edges refer to those for which it is not known where the dependence should be placed before execution reaches that point, occurring because of resource constraints. For these cases, we add to the graph model some additional logic to compute resource usage. The implication is that static edges can be placed independently of any other edge. Most edges in this

Edge	Constraint	Latency
$F \rightarrow D$	Frontend Pipeline Latency	Fixed Lat.
$D \rightarrow R$	Dispatch Before Ready	0
$R \rightarrow E$	Ready Before Execute	0
$E \rightarrow P$	Execution Latency	Fixed, based on Op.
$E \rightarrow P$	Load Latency	Recorded Load Lat.
$P \rightarrow C$	Complete before Commit	Fixed Lat.
$C \rightarrow W$	Commit Before Writeback	Recorded Store Lat.

Table 7: Prism Instruction Nodes and Edges

Static Edge	Constraint	Latency
$F_{i-1} \rightarrow F_i$	Fetch In-order	0
$D_{i-1} \rightarrow D_i$	Dispatch In-order	0
$C_{i-1} \rightarrow C_i$	Commit In-order	0
$F_{i-fw} \rightarrow F_i$	$fw =$ Fetch Width	1
$D_{i-dw} \rightarrow D_i$	$dw =$ Dispatch Width	1
$C_{i-cw} \rightarrow C_i$	$cw =$ Commit Width	1
$F_{i-1} \rightarrow F_i$	Icache Latency	Recorded
$D_{i-ibuf\_sz} \rightarrow F_i$	$ibuf\_sz =$ Frontend Pipeline Buffer Size	0
$C_{i-rob\_sz} \rightarrow D_i$	$rob\_sz =$ Reorder Buffer Size	0
$P_i \rightarrow R_j$	Data Dependence, apply if Inst_j depends on Inst_i	0
$P_i \rightarrow R_j$	Mem Dependence, apply if Inst_j mem depends on Inst_i	0
$P_i \rightarrow D_j$	LQ Size, apply if Inst_j is lq_size loads after Inst_i	0
$P_i \rightarrow D_j$	SQ Size, apply if Inst_j is sq_size stores after Inst_i	0
$C_i \rightarrow F_{i+1}$	Inst_i is serializing	0
$W_i \rightarrow R_j$	Inst_j is non-speculative, and Inst_i is the previous store	0
$C_i \rightarrow F_{i+1}$	Ctrl Inst_i mispredicted, Inst_{i+1} waits until rob is flushed.	Estimated
$P_i \rightarrow P_j$	Load Inst_i pulls cache line for Load Inst_j.	0
Dynamic Edge	**Constraint & Latency**	
$E_i \rightarrow D_j$	IW Size, apply if Inst_i blocks Inst_j from the IW	1
$E_i \rightarrow E_j$	Execution Resource Conflict	1
$P_i \rightarrow F_j$	Load Inst_j blocks in the cache b/c of Inst_i, requires pipe flush.	1
$W_i \rightarrow W_j$	L1 Cache B/W, apply if W_j can enter L1 after W_i	1

Table 8: Static and Dynamic Inter-Instruction Edges

category are similar to previous works, but Prism includes a few additional edge types. Because we added a fetch node, Prism includes an additional edge to model frontend pipeline buffering. Also, we implemented edges to constrain the load and store queues, and for the non-speculative and serializing instructions.

5.6 Prism Validation

Validation Methodology The benchmarks we use in this study come from a combination of benchmarks used in the design evaluations from each previous work. They are listed in full in Table 5(on page 25). For validating the accelerators, we chose processor configurations which attempted to best match descriptions in respective publications. BERET and C-Cores used a dual-issue, inorder CPU, while the quad-issue DySER model used a quad-issue OOO CPU [27, 30, 57]. For validation with SIMD, we set the vector width at 4 words. Table 9 presents a summary of our validation benchmarks and results.

OOO-Core Validation Results To determine if we are over-fitting the OOO core to our simulator, we perform a "cross validation" test: we generate a trace based on the 1-Wide OOO core, and use it to predict the performance/energy of an 8-Wide OOO core, and vice-versa. Figures 17(a) and 17(b) show the results

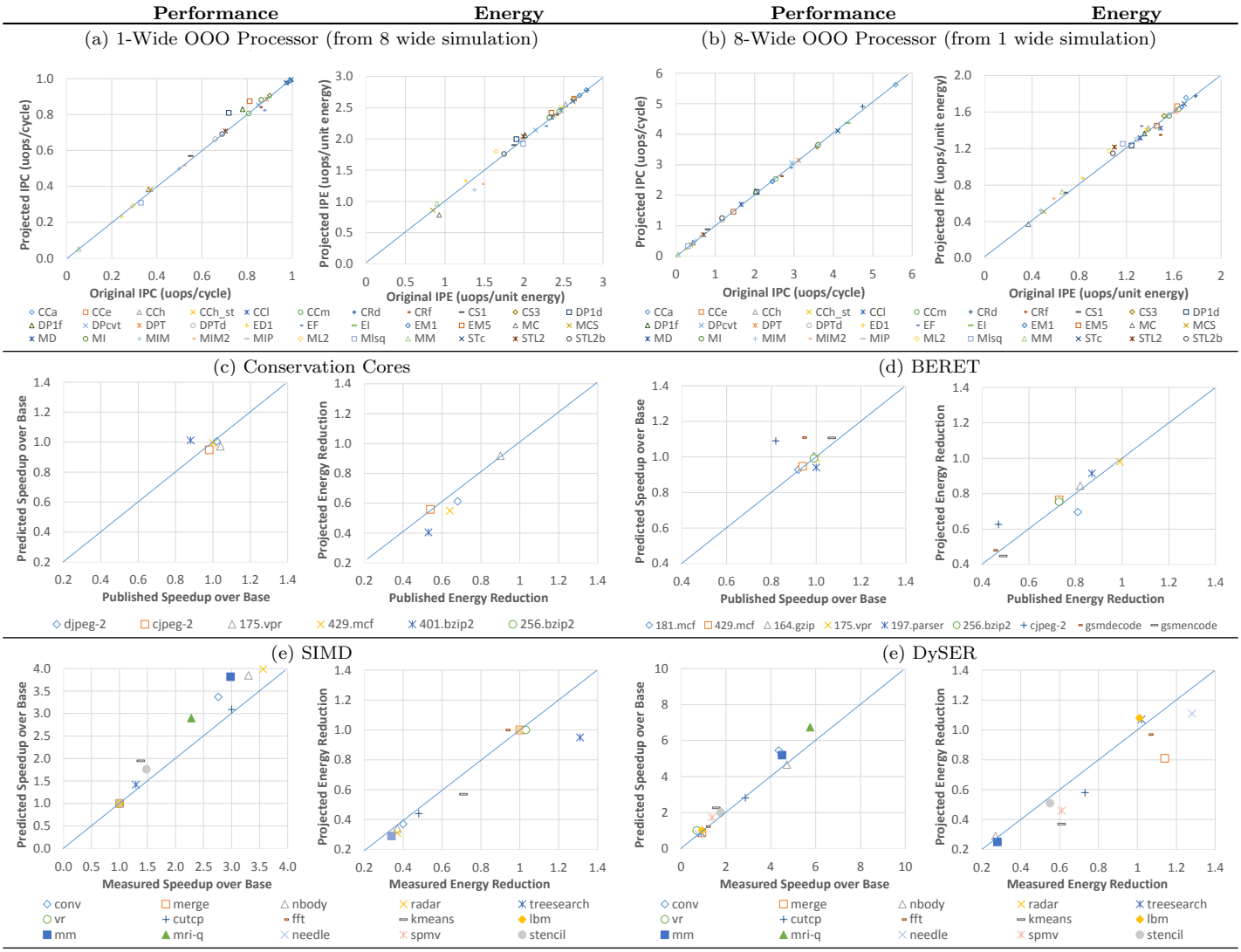


Figure 17: Prism Validation

respectively. The benchmarks are an extension of those used to validate the Alpha 21264[14]. The high accuracy here ($< 4\%$ on average) demonstrates the flexibility of the model in either speeding up or slowing down the execution.

Accelerator Validation Results Figures 17(c-e) show the accelerator validation results. Overall, for every accelerator, we achieve an average absolute error of less than 15%, both in terms of performance and energy reduction, compared to simulator or published data. Note that we do not have access to the NPU energy model, so no energy validation is reported.

Accel.	Benchmarks	Core Config	P	E
8-Wide→1-Wide	μbench[14]	1-Wide OOO	3%	4%
1-Wide→8-Wide	μbench[14]	8-Wide OOO	2%	3%
Conservation Cores	Data/Bench from [56]	2-Wide Inorder	5%	10%
BERET	Data/Bench from [30]	2-Wide Inorder	8%	7%
SIMD	Data/Bench from [29]	4-Wide OOO	12%	7%
DySER	Data/Bench from [29]	4-Wide OOO	15%	15%
NPU	Data/Bench from [17]	4-Wide OOO	11%	

Table 9: Validation Results, P: Avg Perf. Err, E: Avg En. Err

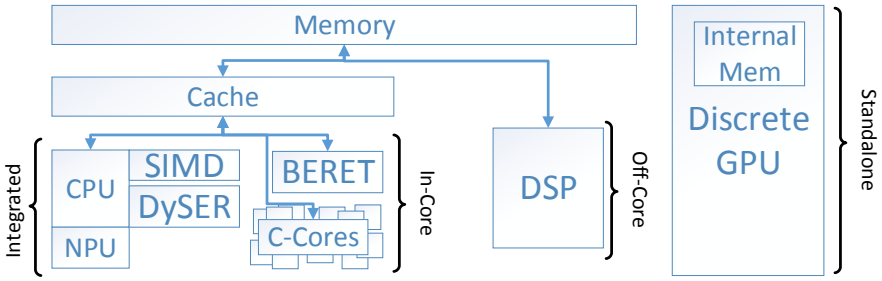


Figure 18: Accelerator Coupling Classification

5.7 Scope of the Transformable Dependence Graph

Here we describe the limits of the TDG model in terms of coupling with the general purpose core and algorithmic change.

Coupling The first dimension which we consider is the scope of is the coupling between the accelerator and the general purpose processor. Figure 18 outlines some examples, and four simple categories. First are accelerators like SIMD or DySER, which are tightly coupled to the GPP, which we call "integrated". They do not have their own interface to the memory system, and rely on the GPP. In-core accelerators are those which completely eschew the GPP, but still share the same memory port. Off-core accelerators are those which attach anywhere else inside the same memory system as the GPP, and standalone accelerators don't require the GPP's memory system at all.

In terms of coupling, we've already shown how in-core and integrated accelerators can be modeled effectively. Off-core and standalone accelerator's pose a significant challenge, because it's difficult to determine how the change in the interface to memory will affect the latency and bandwidth. If the interface to memory is simple, say that no caches are used, and the latency is fixed, then modeling these types of architectures would be possible. One second order error with this approach is that the offloading of certain computations could affect what gets pulled into cache, and therefore the modeling of the GPP's memory accesses would be somewhat inaccurate if the accelerator and GPP share memory at a fine granularity.

A tractable solution to the problem would be to attach a memory system simulator to the TDG model.

This is largely unsatisfying, though, as it would lessen the degree of "purity" of the TDG's graph based model, which is arguably it's foremost strength. Another potential direction is to use statistical techniques to stochastically model the memory system interaction, but this would likely introduce a significant source of error. Overall, it's an open question how effective the TDG could be for modeling off-core and standalone accelerators.

Algorithm Certain accelerators, like GPGPU's, require fundamental changes to the algorithm of the program to be effectively employed. For example, an effective GPU histogram bears very little resemblance to its simplistic CPU counterpart. Unless this algorithmic change can be modeled as a graph transformation, then the TDG cannot model this architecture. One example of program modifications that can be modeled is that of loop vectorization, which can be done because it's straightforward to coalesce loop iterations in the TDG. However, the broader class of algorithmically different accelerators is challenging, and this is why we posit that accurate TDG models for their architectures, including GPUs, will remain elusive.

Interestingly, for certain classes of accelerators, like the NPU, some degree of modeling is possible. This is because we can determine through external methods what the accelerator's internal behavior is.

Computation/Data Specific Optimizations Finally, certain classes of accelerators take advantage of the specific computations and data in order to simplify the hardware. One prominent example of this is to reduce the data-width (or bit-width) to be a size just large enough to fit the data. This could potentially result in significant energy savings. This type of optimization could be capture by the TDG if we added value tracking, and developed an energy model for specialized operations. Though modeling computation/data specific optimizations could be useful, we have not pursed them further as no accelerator's we've evaluated so far has required them.

6 Summary

This dissertation will address the challenges of rising dark silicon and the end of Dennard scaling through researching methods for understanding, developing, and employing accelerators. To enable this, completed work has developed the transformable dependence graph (TDG), which can model acceleration at a higher level of abstraction while still capturing important interactions with the general purpose processor and the applications. The research will use this abstraction in three areas: First, to decipher accelerators' fundamental limitations and opportunities, which could help accelerator designers focus their efforts. Second, to enable accelerator design practices which are fast and explore a broad design space. And third, to enable

the study of practical techniques for scheduling multi-accelerator systems.

The three pieces of prosed work are far from independent. Part of the effectiveness of the TDG-based design strategy comes from the ability to perform limit studies. By enabling multi-accelerator scheduling on applications, we may open new limits and opportunities for acceleration. More concretely, the NLA design we are proposing can be viewed as breaking the limitations of the BERET architecture, and the architectures we develop using our design strategy will help to make multi-acceleration useful. Taking the components of the proposed work together, the goal of the proposed research is to enable the development of accelerator systems which can provide order-of-magnitude better performance and energy-efficiency.

References

[1] *Parboil Benchmark Suite*. http://impact.crhc.illinois.edu/parboil.php.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate Vs. IPC : The End of the Road for Conventional Microprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[3] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 26–36, New York, NY, USA, 2010. ACM.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[6] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.

[7] C. Caşcaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik. A taxonomy of accelerator architectures and their programming models. *IBM J. Res. Dev.*, 54(5):473–482, Sept. 2010.

[8] A. A. Chien, A. Snavely, and M. Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. In *ICCS*, 2011.

[9] G. Chrysos and S. P. Engineer. Intel® xeon phi coprocessor (codename knights corner). In *Proceedings of the 24th Hot Chips Symposium, HC*, 2012.

[10] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPUs? In *MICRO*, 2010.

[11] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *ISCA '08*, pages 389 –400.

[12] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 30–40, Washington, DC, USA, 2004. IEEE Computer Society.

[13] N. Clark, H. Zhong, K. Flautner, K. Fan, S. Mahlke, and K. V. Nieuwenhove. Optimode: Programmable accelerator engines through retargetable customization. In *Hot Chips 16*, Aug 2004.

[14] R. Desikan, D. Burger, and S. W. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, July 2001.

[15] L. Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 5(1):1–145, 2010.

[16] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA '11*.

[17] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[18] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27(2):3:1–3:37, May 2009.

[19] B. Fields, R. Bodik, M. Hill, and C. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 228–239, 2003.

[20] B. Fields, R. Bodík, and M. D. Hill. Slack: maximizing performance under technological constraints. In *ISCA '02*, pages 47–58.

[21] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[22] D. Fischer, J. Teich, R. Weper, and M. Thies. Buildabong: A framework for architecture/compiler co-exploration for asips. *Journal of Circuits, Systems and Computers*, 12(03):353–375, 2003.

[23] M. Galanis, G. Theodoridis, S. Tragoudas, and C. Goutis. A high-performance data path for synthesizing dsp kernels. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(6):1154–1162, 2006.

[24] C. Galuzzi and K. Bertels. The instruction-set extension problem: A survey. *ACM Trans. Reconfigurable Technol. Syst.*, 4(2):18:1–18:28, May 2011.

[25] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[26] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, Mar. 2000.

[27] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy efficient computing. *IEEE Micro*, 33(5), 2012.

[28] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA 2011*.

[29] V. Govindaraju, T. Nowatzki, and K. Sankaralingam. Breaking simd shackles with an exposed flexible microarchitecture and the access execute pdg. In *PACT*, pages 341–351, 2013.

[30] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO '11*.

[31] 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*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[32] M. Hempstead, G.-Y. Wei, and D. Brooks. Navigo: An early-stage model to study power-contrained architectures and specialization. In *Proceedings of Workshop on Modeling, Benchmarking, and Simulations (MoBS)*, 2009.

[33] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with Lisa*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[34] S. Hong and H. Kim. An integrated gpu power and performance model. In *ISCA '10*.

[35] R. Iyer. Accelerator-rich architectures: Implications, opportunities and challenges. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 106–107, 2012.

[36] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 330–335, 1997.

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO '09*.

[38] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Soda: A low-power architecture for software radio. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 89–101, Washington, DC, USA, 2006. IEEE Computer Society.

[39] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. MICRO-45. IEEE Computer Society, 2012.

[40] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *Design Test of Computers, IEEE*, 26(4):18–25, July 2009.

[41] J. Meng, V. Morozov, K. Kumaran, V. Vishwanath, and T. Uram. Grophecy: Gpu performance projection from cpu code skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 14. ACM, 2011.

[42] M. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole. Modeling and predicting application performance on hardware accelerators. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 73–73, 2011.

[43] S. Navada, N. K. Choudhary, S. V. Wadhavkar, and E. Rotenberg. A unified view of non-monotonic core selection and application steering in heterogeneous chip multiprocessors. PACT '13, 2013.

[44] R. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, 2004.

[45] T. Nowatzki, M. Ferris, K. Sankaralingam, C. Estan, N. Vaish, and D. A. Wood. *Optimization and Mathematical Modeling in Computer Architecture*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[46] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *PLDI '13,* **To Appear**.

[47] C. Nugteren and H. Corporaal. A modular and parameterisable classification of algorithms. Technical Report ESR-2011-02, Eindhoven University of Technology, 2011.

[48] C. Nugteren and H. Corporaal. The boat hull model: adapting the roofline model to enable performance prediction for parallel computing. In *PPOPP '12*, pages 291–292, 2012.

[49] S. Onder and R. Gupta. Automatic generation of microarchitecture simulators. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 80–89, May 1998.

[50] S. Padmanabha, A. Lukefahr, R. Das, and S. A. Mahlke. Trace based phase prediction for tightly-coupled heterogeneous cores. In *MICRO*. ACM, 2013.

[51] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: Balancing efficiency & flexibility in specialized computing. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 24–35, New York, NY, USA, 2013. ACM.

[52] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[53] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 11–22, New York, NY, USA, 2012. ACM.

[54] M. Vachharajani, N. Vachharajani, and D. I. August. The liberty structural specification language: A high-level modeling language for component reuse. PLDI '04, 2004.

[55] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August. The liberty simulation environment: A deliberate approach to high-level system modeling. *ACM Transactions on Computer Systems (TOCS)*, 24(3):211–249, 2006.

[56] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *ASPLOS '10*.

[57] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS XV*.

[58] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188, New York, NY, USA, 1991. ACM.

[59] G. Wang, D. Chen, J. Chen, J. Ma, and T. Chen. A performance model for run-time reconfigurable hardware accelerator. In *Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies*, APPT '09, pages 54–66, Berlin, Heidelberg, 2009. Springer-Verlag.

[60] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[61] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 249–260, New York, NY, USA, 2013. ACM.

[62] T. Zidenberg, I. Keslassy, and U. Weiser. Optimal resource allocation with multiamdahl. *Computer*, 46(7):70–77, 2013.