Analyzing Behavior Specialized Acceleration

Tony Nowatzki Karthikeyan Sankaralingam
University of Wisconsin - Madison
{tjn,karu}@cs.wisc.edu

Abstract

Hardware specialization has become a promising paradigm for overcoming the inefficiencies of general purpose microprocessors. Of significant interest are Behavioral Specialized Accelerators (BSAs), which are designed to efficiently execute code with only certain properties, but remain largely configurable or programmable. The most important strength of BSAs – their ability to target a wide variety of codes – also makes their interactions and analysis complex, raising the following questions: can multiple BSAs be composed synergistically, what are their interactions with the general purpose core, and what combinations favor which workloads? From a methodological standpoint, BSAs are also challenging, as they each require ISA development, compiler and assembler extensions, and either simulator or RTL models.

To study the potential of BSAs, we propose a novel modeling technique called the Transformable Dependence Graph (TDG) - a higher level alternative to the time-consuming traditional compiler+simulator approach, while still enabling detailed microarchitectural models for both general cores and accelerators. We then propose a multi-BSA organization, called ExoCore, which we model and study using the TDG. A design space exploration reveals that an ExoCore organization can push designs beyond the established energy-performance frontiers for general purpose cores. For example, a 2-wide OOO processor with three BSAs matches the performance of a conventional 6-wide OOO core, has 40% lower area, and is 2.6× more energy efficient.

1. Introduction

Hardware specialization has become a promising paradigm for continued efficiency improvements. The insight of this paradigm is that, depending on the type of program or code, relaxing certain capabilities of the general purpose core, while augmenting it with others, can eliminate energy overheads and greatly improve performance.

ASPLOS '16, April 02 - 06, 2016, Atlanta, GA, USA

ACM 978-1-4503-4091-5/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2872362.2872412

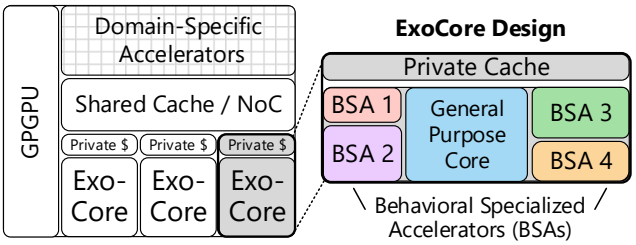

Figure 1: ExoCore-Enabled Heterogeneous System

Within the broad specialization paradigm, approaches can be categorized as either domain specialized or behavior specialized. A domain specialized approach uses hardware customized for a specific domain of computation or even a single algorithm (resembling the SoC paradigm). This allows these accelerators to be extremely efficient for certain domains. Since this approach requires domain-specific information to be conveyed to the architecture, their use is largely non-transparent to the programmer. Examples include accelerators for convolution [42], neural networks [7, 30], approximate computing [3, 13], video encoding [19] and database processing [24, 56].

Behavior specialized accelerators (BSAs), the focus of this work, differ in that they attempt to exploit program behaviors and their inter-relationship to hardware microarchitecture[1]. A well-known example is short-vector SIMD instructions, which exploit local data-parallelism. Research examples include BERET [18], which specializes for hot program-traces, XLOOPS [49], which specializes for loop dependence patterns and DySER [17], which exploits memory/computation separability in data-parallel code. Specializing for program behaviors is advantageous both because fewer accelerators can target a large variety of codes, and because these behaviors are typically analyzable by a compiler, meaning their use can be largely transparent to the programmer. Though non-transparent BSAs have been proposed (eg. LSSD [37]), we focus on transparent BSAs in this work.

Since programs exhibit a variety of behaviors, and BSAs each target different behaviors, it is natural to consider combining them. We propose an organization called ExoCore, as shown in Figure 1. An ExoCore integrates a general purpose core with several BSAs targeted at different program behav-

[1] The line between BSA and DSA can be sometimes blurry. Considering the NPU [13] accelerator, a sophisticated compiler could automatically determine neural-network mappable regions, perhaps making NPU a BSA.

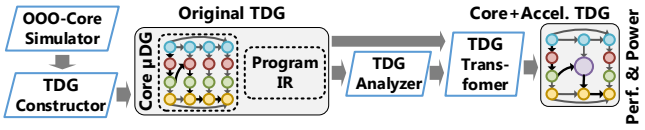

Figure 2: TDG-based Accelerator Modeling Framework

iors. At run-time, depending on the affinity of the code, the execution may migrate to one of the BSAs, a process which is transparent to the programmer. To enable fast switching between cores, without the need for migrating state, BSAs share the virtual memory system and cache hierarchy.

Studying Behavioral Accelerators Considering designs that incorporate multiple synergistic behavior-specialized accelerators, many questions become immediately apparent: can the performance and energy benefits be worth the area costs? Is there opportunity for multiple-BSAs inside applications, and at what granularity? What types of BSAs are synergistic, and is there a minimal useful set? Which type of general purpose core should they integrate with? When accelerators target overlapping regions, how should the decision be made to switch between BSAs?

We argue that thus far, these questions have not been addressed adequately for at least two reasons. First, prior to the advent of "dark silicon", such a design would not have been sensible given that certain portions of the core would go unused at any given time – now the tradeoffs are more plausible. Secondly, we simply lack effective ways to study an organization like ExoCore. Evaluating even one BSA requires extensive effort, including the definition of a new ISA and possibly ISA extensions for the host processor, compiler analysis, transformations and assembler implementation, and finally for evaluation requires either a simulator or RTL model. Exploring ExoCore designs incorporating many BSAs has so far been intractable, and evaluating new BSAs takes considerable effort; a higher-level evaluation methodology, retaining the ability of modeling the effect of *both* the compiler and the detailed interaction of the microarchitecture seems necessary.

Goals and Contributions This paper's goal is to elucidate the potentials of synergistic BSAs. To achieve this, the first thrust of this paper is to propose a novel BSA-modeling methodology, called the transformable dependence graph (TDG). The approach is to combine semantic information about program properties (eg. loop dependence, memory access patterns) and low-level hardware events (eg. cache misses, branch mis-predictions, resource hazards) in a single trace-based dependence graph representation. This dependence graph is transformed, by adding or removing edges, to model how it specializes for certain program behaviors (Figure 2).

Using the TDG as the modeling methodology, we explore a design space of ExoCore organizations that draws accelerators from the literature, evaluating 64 design points across more than 40 benchmarks. Our analysis shows that

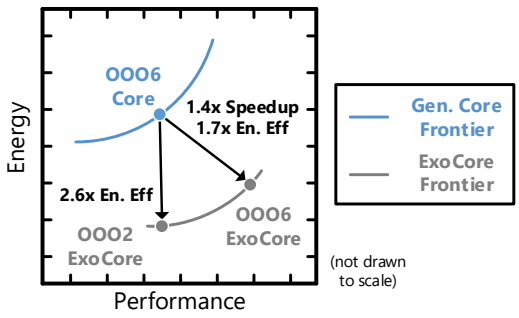

Figure 3: Results of Design-Space Exploration Across 64 ExoCore Configurations

having multiple BSAs can be helpful both within workload domains and within single applications. Understanding this design space would have been intractable without a TDG style approach[2].

To summarize, this paper's contributions are:

- A novel methodology (TDG) for modeling behavior specialized acceleration (see `www.cs.wisc.edu/vertical/tdg` for implementation details), that is practical and accurate (less than 15% error in relative performance and energy efficiency for existing BSAs).
- Defining the ExoCore concept, and developing a synergistic composition of BSAs which significantly pushes the performance and energy frontiers of traditional cores.
- Design-space exploration of ExoCore configurations (Figure 3). Specifically, a 2-wide OOO processor with three BSAs matches the performance of a conventional 6-wide OOO core with SIMD, has 40% lower area and is 2.6× more energy efficient. We also find that BSAs have synergistic opportunities across and *within* workloads.

Paper Organization We first describe the background, intuition, and details of the TDG-based accelerator modeling approach, along with single-accelerator validation (Section 2). We then use the TDG methodology to design and model an ExoCore system with four synergistic BSAs (Section 3). Section 4 has methodology, and Section 5 has results. Section 6 is related work, and Section 7 concludes.

2. Transformable Dependence Graphs

In this section, we describe our approach for modeling behavior specialized accelerators: the Transformable Dependence Graph (TDG). We first describe the requirements for a BSA-modeling framework, then overview our proposed approach, and go into detail with a specific accelerator example. We also discuss implementation, validation, usage and limitations.

[2] We note that TDG is a simulation-modeling technique that does not produce a compiler artifact, or microarchitecture spec, or RTL of the modeled BSA. Its potential usefulness is in research of accelerator ideas and early design-space exploration - it is analogous to a cycle-level trace simulator.

2.1 Insights and Background

Requirements For modeling the workings of behavior specialized accelerators, a useful evaluation approach must capture the following aspects of execution:

1. **General Purpose Core Interaction** Since transparent accelerators selectively target certain code, the general purpose core must be modeled in detail (eg. pipelining, structural hazards, memory-system etc.). In addition, accelerators can interact with different parts of the general purpose core (eg. sharing the LSQ), so this must be modeled as well.
2. **Application/Compiler Interaction** Much of a BSA's success depends on how well the compiler can extract information from the application, both for determining a valid accelerator configuration, and for deciding which accelerator is appropriate for a given region.
3. **Accelerator Behavior** Naturally, the low-level detailed workings of the accelerator must be taken into-account, including the microarchitecture and dynamic aspects of the execution like memory latency.

A traditional compiler+simulator can capture the above aspects very well, but the effort in building multi-BSA compilers and simulators is time consuming, if not intractable. What we require are higher-level abstractions.

Leveraging Microarchitectural Dependence Graphs (μDGs) For a higher-level model of microarchitectural execution, we turn to the μDG, a trace-based representation of a program. It is composed of nodes for microarchitectural events and edges for their dependences. It is constructed using dynamic information from a simulator, and as such, is input-dependent. The μDG has traditionally been used for modeling out-of-order cores [15, 28]. Overall, it offers a detailed-enough abstraction for modeling microarchitecture, yet is still abstract and easy to model modifications/additions of effects.

What is missing from the above is the capability of capturing compiler/application interactions. Our insight is that *graph transformations* on the μDG can capture the effects of behavioral specialization – after all, a BSA simply relaxes certain microarchitectural dependences while adding others. To perform these transformations, we require knowing the correspondence between the program trace and the static program IR. This can be reconstructed from the binary.

2.2 The Transformable Dependence Graph

Approach Overview Putting the above together, the crux of our approach is to build the Transformable Dependence Graph (TDG), which is the combination of the μDG of the OOO core, and a Program IR (typically a standard DFG + CFG) which has a one-to-one mapping with μDG nodes.

As shown in Figure 2, a simulator produces dynamic instruction, dependence, and microarchitectural information, which is used by the constructor to build the TDG. The TDG is analyzed to find acceleratable regions and determine the strategy for acceleration. The TDG-transformer modifies the original TDG, according to a graph re-writing algorithm, to create the combined TDG for the general purpose core and accelerator. As part of the representation, the TDG carries information about overall execution time and energy. The next subsection describes the approach using an example.

Notation To aid exposition, the notation $\text{TDG}_{\text{GPP,ACCEL}}$ refers to a TDG representation of a particular general purpose processor (GPP) and accelerator. $\text{TDG}_{\text{OOO4,SIMD}}$, for example, represents a quad-issue OOO GPP with SIMD. As a special case, the original TDG prior to any transformations (not representing an accelerator) is $\text{TDG}_{\text{GPP},\emptyset}$.

2.3 Transformable Dependence Graph Example

Here we define the components of our approach using a running example in Figure 4, which is for transparently applying a simple fused multiply-accumulate (`fma`) instruction. We intentionally choose an extremely simple example for explanatory purposes, and note how a more complex accelerator would implement that component. Detailed modeling of such accelerators are in Section 3.2.

Constructing the TDG To construct $\text{TDG}_{\text{GPP},\emptyset}$, a conventional OOO GPP simulator (like gem5 [5]) executes an unmodified binary[3], and feeds dynamic information to the TDG constructor (Figure 4(a)). The first responsibility of the tool is to create the original μDG, which embeds dynamic microarchitectural information, including data and memory dependences, energy events, dynamic memory latencies, branch mispredicts and memory addresses. We note that this makes the TDG input dependent, which is similar to other trace-based modeling techniques.

To explain an example, Figure 4(b) shows the μDG for the original OOO core, which in this case was a dual issue OOO. Here, nodes represent pipeline stages, and edges represent dependencies to enforce architectural constraints. For example, edges between alternate dispatch and commit nodes model the width of the processor ($D_{i-2} \xrightarrow{1} D_i$, $C_{i-2} \xrightarrow{1} C_i$). The FU or memory latency is represented by edges from execute to complete ($E_i \rightarrow P_i$), and data dependencies by edges between complete to execute ($P_i \xrightarrow{0} E_j$).

The second responsibility of the constructor is to create a program IR (also in Figure 4(b)) where each node in the μDG has a direct mapping with its corresponding static instruction in the IR. We analyze the stream of instructions from the simulator, using known techniques to reconstruct the CFG, DFG with phi-information, and loop nest structure using straightforward or known techniques [33]. Also, register spill and constant access is identified for later optimization. To account for not-taken control paths in the program,

[3] Our implementation assumes that compiler auto-vectorization is off.

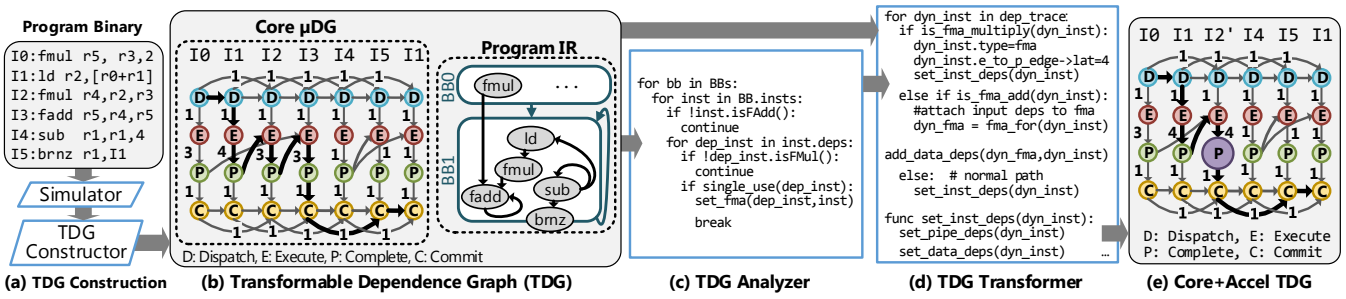

Figure 4: Example TDG-Based model for transparent fused multiply-add (`fma`) specialization.

we augment the program IR with the CFG from binary analysis.

TDG Analyzer The next step is to analyze the TDG to determine which program regions can be the legally and profitably accelerated, as well as the "plan" for transformation. This "plan" represents the modifications a compiler would make to the original program. We explain with our example.

Figure 4(c) shows the algorithm (in pseudo-code) required for determining the `fma` instructions. To explain, the routine iterates over instructions inside a basic block, looking for a `fadd` instruction with a dependent `fmul`, where the `fmul` has a single use. The function `set_fma(inst1,inst2)` records which instructions are to be accelerated, and passes this "plan" to the TDG transformer. In concrete terms, a TDG-analysis routine is a C++ module that operates over the TDG's trace or IR, and the "plan" is any resulting information or data-structures that are stored alongside the TDG.

While the above is basic block analysis, more complex accelerators typically operate on the loop or function level. For example, determining vectorizability in SIMD would require analyzing the IR for inter-iteration data dependences.

TDG Transformer This component transforms the original TDG to model the behavior of the core and accelerator according to the plan produced in the previous step. It applies accelerator-specific graph transformations, which are algorithms for rearranging, removing, and reconstructing μDG nodes and dependence edges. In our notation, this is the transformation from $\text{TDG}_{\text{GPP-X},\emptyset}$ to $\text{TDG}_{\text{GPP-Y,ACCEL}}$.

Figure 4(d) outlines the algorithm for applying the `fma` instruction, which iterates over each dynamic instruction in the μDG. If it is an accelerated `fmul`, it changes its type to `fma` and updates its latency. If the original instruction is an accelerated `fadd`, it is elided, and the incoming data dependences are added to the associated `fma`.

This simple example operates at an instruction level, but of course more complex accelerators require modifications at a larger granularity. For instance, when vectorizing a loop, the μDG for multiple iterations of the loop can be collected and used to produce the vectorized μDG for a single new iteration.

Core+Accelerator TDG The core+accelerator TDG represents their combined execution, an example of which is in Figure 4(e), for $\text{TDG}_{\text{OOO2,fma}}$. Here, `I2'` represents the specialized instruction, and `I3` has been eliminated from the graph. In practice, more complex accelerators require more substantial graph modifications. One common paradigm is to fully switch between a core and accelerator model of execution at loop entry points or function calls.

Finally, this TDG can be analyzed for performance and power/energy. The length of the critical path, shown in bold in the figure, determines the execution time in cycles. For energy, we associate events with nodes and edges, which can be accumulated and fed to standard energy-modeling tools.

Accel.	Base	P Err.	P Range	E Err.	E Range
OOO8→1	–	3%	0.05→1.0 IPC	4%	0.75→2.75 IPE
OOO1→8	–	2%	0.02→5.5 IPC	3%	0.39→1.7 IPE
C-Cores	IO2	5%	0.84→1.2×	10%	0.5→0.9×
BERET	IO2	8%	0.82→1.17×	7%	0.46→0.99×
SIMD	OOO4	12%	1.0→3.6×	7%	0.30→1.3×
DySER	OOO4	15%	0.8→5.8×	15%	0.25→1.28×

Table 1: Validation Results (P: Perf, E: Energy)

2.4 Implementation: Prism

Our framework's implementation, *Prism*, generates the original TDG using gem5 [5], enabling analysis of arbitrary programs. We implement custom libraries for TDG generation, analysis and transformation (more details on specific modeling edges are in a related paper [38]). Since transforming multi-million instruction traces can be inefficient, Prism uses a windowed approach. Windows are large enough to capture specialization granularity (max ∼10000 instructions). The final outputs include cycle count and average power.

Power and Area Estimation Prism accumulates energy event counts for both the GPP and accelerator from the TDG. It then uses McPAT [29] internally for computing power, calling McPAT routines at intervals over the program's execution. We use 22nm technology. The GPP core activity counts are fed to McPAT [29], a state-of-the-art GPP power model. For accelerators, a combination of McPAT (for FUs) and CACTI [34] is used, and for accelerator-specific hardware we use energy estimates from existing publications.

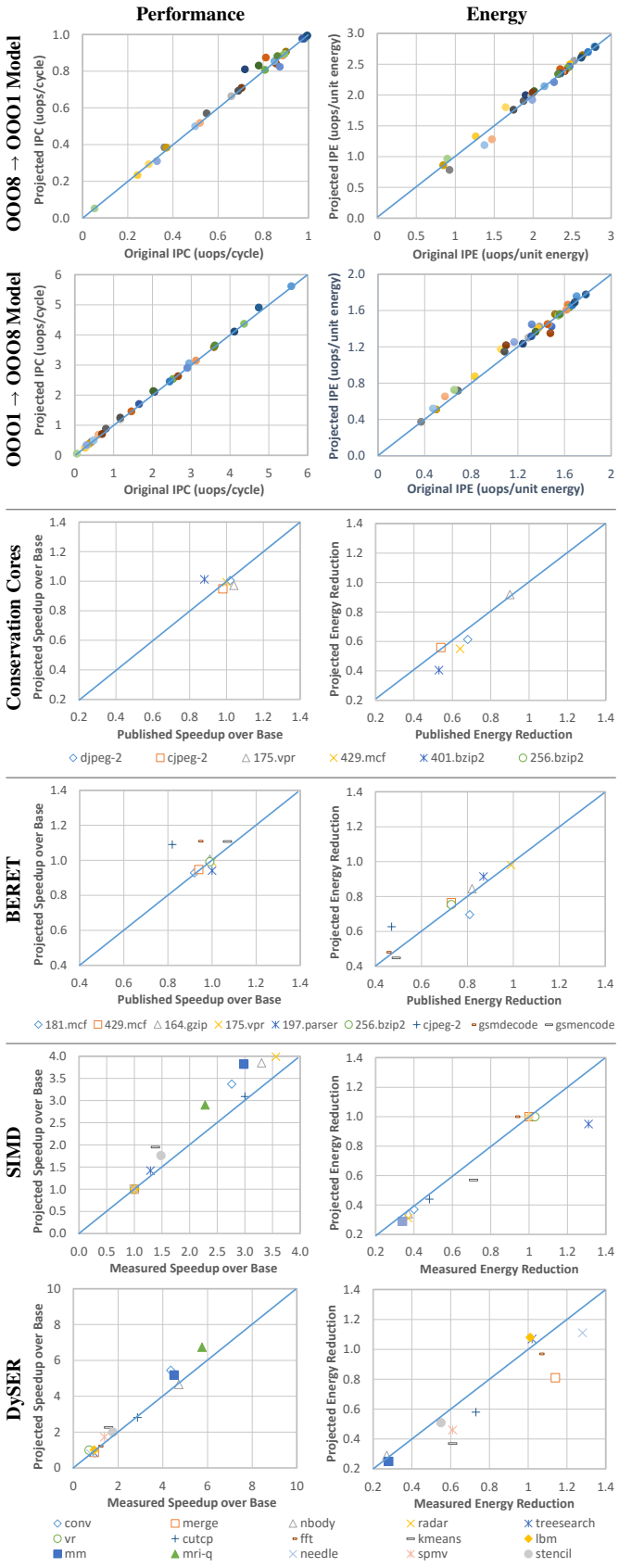

Figure 5: Prism Validation

2.5 Core and Single-Accelerator Validation

We perform validation by comparing the TDG against the published results of four accelerators, each of which use traditional compiler+simulator evaluation. These results used benchmarks from the original publications [17, 18, 53], and Parboil [1] and Intel's microbenchmarks for SIMD. The original μDG is generated by fast-forwarding past initialization, then recording for 200 million instructions. Here, the error is calculated by comparing the relative speedup and energy benefits over a common baseline, where we configured our baseline GPP to be similar to the reported baseline GPP. Note that for SIMD and DySER, we were able to adjust the transformations to match the compiler output, as we had access to these compiler infrastructures.

Table 1 presents a validation summary for the OOO core and accelerators, where column "Base" is the baseline GPP, "Err." is the average error, and "Range" is the range of compared metrics. Figure 5 shows the validation graphs for each architecture, including performance (left) and energy (right). Each point represents the execution of one workload, and the distance to the unit line represents error. The X-Axis is the validation target's estimate (from published results or measured from simulation), and the Y-Axis is the TDG's estimate of that metric. We describe the validation of the core and accelerators below.

OOO-Core 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. The first two graphs in Figures 5 show the results. The benchmarks in this study [2] are an extension of those used to validate the Alpha 21264 [9] (we omit names for space). The high accuracy here ($< 4\%$ on average) demonstrates the flexibility of the model in either speeding up or slowing down the execution.

Conservation Cores are automatically generated, simple hardware implementations of application code [53], which are meant as offload engines for in-order cores. We validate the five benchmarks in the above paper, and achieve an average of 5% and 10% average error in terms of performance improvement and energy reduction. The worst case is for 401.bzip2, where we under-predict performance by 15%.

BERET is also an offload engine for hot loop traces, where only the most frequently executed control path is run on the accelerator, and diverging iterations are re-executed on the main processor [18]. Efficiency is attained through serialized execution of compound functional units. We achieve an average error of 8% and 7% in terms of performance improvement and energy reduction. We over-predict performance slightly on cjpeg and gsmdecode, likely because we approximate using size-based compound functional units, rather than more restrictive pattern based (to not over-conform to the workload set).

BSA (**Acronym**)	Exploited App. Behavior	Benefits vs General Core	Drawbacks vs General Core	Granularity	Inspired By
Short-Vector **SIMD**	Data-parallel loops with little control	Fewer instructions and less port contention	Masking/predicated inst penalty	Inner Loops	
Data Parallel CGRA (**DP-CGRA**)	Parallel loops w/ separable compute/memory	Vectorization + fewer insts. on general core	Extra comm. insts, predicated inst penalty	Inner Loops	DySER [17], Morphosys [47]
Non-speculative Dataflow (**NS-DF**)	Regions with non-critical control	Cheap issue width, larger instruction window	Lacks control speculation, requires instr. locality	Nested Loops	SEED [36], Wavescalar [50]
Trace-Speculative Proc. (**Trace-P**)	Loops w/ consistent control (hot traces)	Similar to above, but larger compound insts.	Trace mispeculation requires re-execution	Inner Loop Traces	BERET [18], CCA [8]

Table 2: Tradeoffs of Behavior Specialized Accelerators (BSAs) in this Work

SIMD validation is performed using the Gem5 Simulator's implementation, configured as a 4-Wide OOO processor. Figure 5(e) shows how we attain an average error of 12% and 7% in terms of performance improvement and energy reduction. Our predictions for SIMD are intentionally optimistic, as there is evidence that compilers will continue to see improved SIMD performance as their analysis gets more sophisticated. We remark that our transformations only consider a straight-forward auto-vectorization, and will not be accurate if the compiler performs data-layout transformations or advanced transformations like loop-interchange.

DySER is a coarse grain reconfigurable accelerator (CGRA), operating in an access-execute fashion with the GPP through a vector interface [17]. On the Parboil and Intel microbenchmarks, we attain an average error of 15% for both speedup and energy reduction.

In summary, The TDG achieves an average error of less than 15% for estimating speedup and energy reduction, compared to simulator or published data.

2.6 Using TDG Models in Practice

The TDG can be used to study new BSAs, their compiler interactions and the effect of varying input sets. In practice, running TDG models first requires TDG-generation through a conventional simulator. The generated TDG can be used to explore various core and accelerator configurations. Since the TDG is input-dependent, studying different inputs requires the re-running the original simulation.

Implementing a TDG model is a process of writing IR analysis routines, graph-transformation algorithms and heuristics for scheduling, as outlined in Appendix A.

2.7 Limitations of TDG Modeling

Lack of Compiler Information First, since the TDG starts from a binary-representation, it lacks native compiler information, which sometimes must be approximated in some way. An example is memory aliasing between loop instructions, useful for determining vectorization legality. In such cases, we use dynamic information from the trace to estimate these features, though of course this is optimistic.

Other Sources of Error Another consequence of beginning from a binary representation are ISA artifacts in the TDG. One important example is register spilling. In this case, the TDG includes a best-effort approach to identify loads and stores associated with register spills, which can potentially be bypassed in accelerator transformations.

The graph representation is itself constraining, in particular for modeling resource contention. To get around this, we keep a windowed cycle-indexed data structure to record which TDG node "holds" which resource. The consequence is that resources are preferentially given in instruction order, which may not always reflect the microarchitecture.

Another source of error is unimplemented or abstracted architectural/compiler features. An example from this work is the lack of a DySER spatial scheduler – the latency between FUs is estimated. Of course, TDG models can be made more detailed with more effort.

Flexibility The μDG itself embeds some information about the microarchitectural execution (eg. memory latency, branch prediction results), meaning that its not possible to change parameters that affect this information without also re-running the original simulation. Understanding how those components interact with specialization would require recording multiple TDGs.

Transformation Expressiveness Some transformations are difficult to express in the TDG, limiting what types of architectures/compilation techniques can be modeled. In particular, non-local transforms are challenging, because of the fixed instruction-window that the TDG considers. One example of this would be an arbitrary loop-interchange.

3. Designing ExoCore Systems

Now that we have described an effective modeling technique for studying multiple BSA designs, the TDG, we employ it to study multi-BSA organizations - we call such an architecture an *ExoCore*. As shown earlier in Figure 1, an ExoCore integrates a general purpose core with several other programmable or configurable BSAs targeted at different kinds of program behaviors. At runtime, depending on the

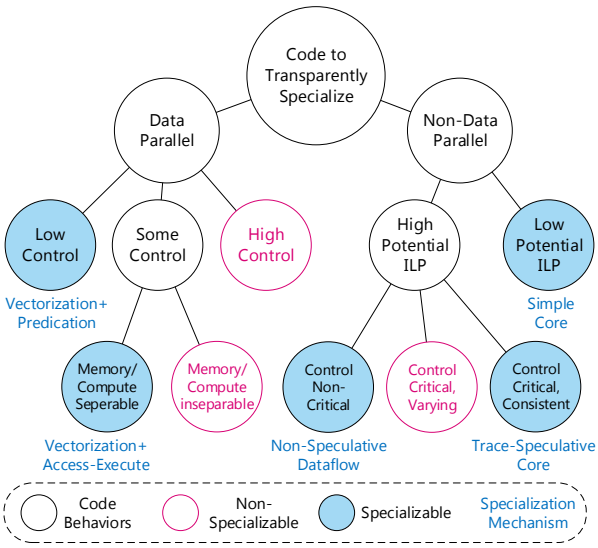

Figure 6: This work's behavior-space.

affinity of the code, the execution may migrate to one of the BSAs, a process which is transparent to the programmer.

In this section, we first discuss how to construct a composition of BSAs to form an ExoCore, then give background on the individual BSAs that comprise it, and finally describe how they are modeled using the TDG.

3.1 Opportunities for Behavioral Specialization

An effective ExoCore design must incorporate accelerators which are *synergistic*, meaning they can effectively work together for added benefits. Intuitively, BSAs that are synergistic will simply specialize different types of behaviors. Figure 6 shows an example taxonomy of program behaviors, where leaf nodes in this tree are program types that can be mapped to specific hardware specialization mechanisms.

With our behavior space outlined, we can now begin to compose a set of BSAs which target those behaviors. Fortunately, we can draw from the existing accelerator literature to fill these niches. Table 2 shows how different accelerators can map onto the specializable behaviors, and gives insights into their benefits and drawbacks versus a general purpose core. We describe these accelerators below; their high-level architecture is in Figure 7.

SIMD Short vector **SIMD** is a well known BSA for targeting data parallel regions, where a limited amount of control and memory irregularity exists.

Data-Parallel Access-Execute Past a certain degree of control, SIMD is no longer effective. An alternate approach is to offload the computation of the loop to a CGRA which natively supports control, and pipeline the CGRA for executing parallel loops (eg. DySER [17], Morphosys [47]). This works well when the memory and computation are separable (fewer communication instructions). Also, these architectures handle some memory irregularity by providing a shuffle network in their flexible input/output interfaces.

We call this design Data-Parallel CGRA (**DP-CGRA**). Its design point has 64 functional units (FUs), and is configured similar to previous proposals [17].

Non-speculative Dataflow In code regions that are not data-parallel, but still have high potential ILP, non-speculative dataflow processors can be highly effective, especially when the control flow does not lie on the critical path.

We model this BSA after the recent SEED [36] architecture, using distributed dataflow units communicating over a bus, and compound FUs (CFUs) for computation. This design targets inlined nested loops with 256 static compound instructions. We refer to this as **NS-DF**, for non-speculative dataflow.

Trace-Speculative Core Often, control is on the critical path, meaning speculation is necessary for good performance, but it is *highly* biased – creating one hot path through a loop. Architectures like BERET [18] exploit this by sequencing through a speculative trace of instructions, using CFUs for energy efficiency. Instructions diverging from the hot loop trace require re-execution on the general core.

We model a trace-speculative BSA similar to BERET, except that we add dataflow execution. This enables the design to be more competitive with an OOO core. We add a loop-iteration versioned store buffer to save speculative iterations. We refer to this design as a **Trace-P** for trace-processor[4]. Compared to NS-DF, Trace-P requires half as much operand storage, and can have larger CFUs, as compound instructions in Trace-P can cross control boundaries.

ExoCore Architecture Organization Putting the above together, Figure 7 shows how an ExoCore combines a general core with the four different BSAs. DP-CGRA and SIMD are integrated with vector datapaths from the general core. All designs besides SIMD are *configurable*, and require a configuration datapath. NS-DF and Trace-P are *offload* architectures, meaning they have their own interfaces to the cache hierarchy, and can power down parts of the core.

We emphasize that the detailed microarchitecture is not the emphasis or contribution of this work. The novel and new ability to model such an architecture using the TDG, and the implications of this design organization are the main contributions.

3.2 TDG-Modeling of BSAs

We now discuss how each BSA is modeled inside the TDG framework. Each accelerator model discusses the analysis plan and transforms, and Figure 8 shows an example code and transform for each BSA.

SIMD (Loop Auto-vectorization) TDG For SIMD we focus on vectorizing independent loop iterations, as this is the most common form of auto-vectorization.

TDG Analysis: First, a pass optimistically analyzes the TDG's memory and data dependences. Memory-dependences

[4] Not to be confused with the Trace Processor from [43]

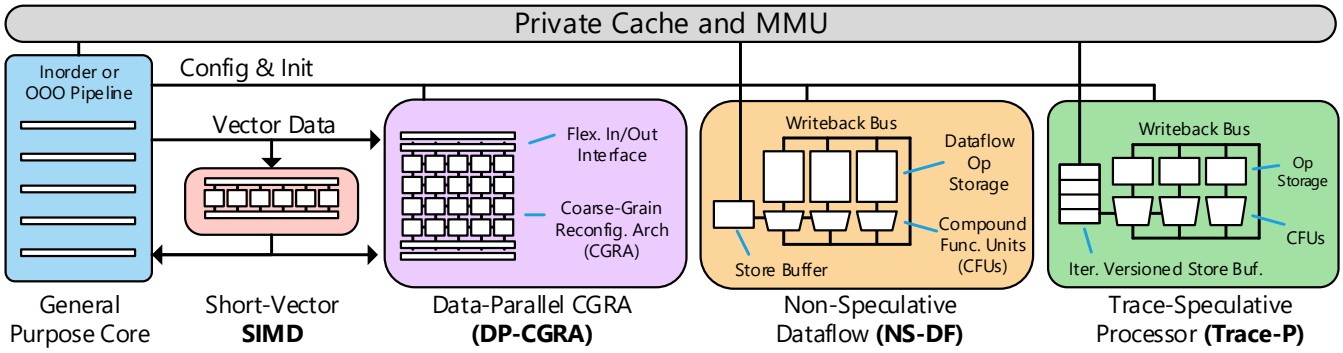

Figure 7: Example ExoCore Architecture Organization

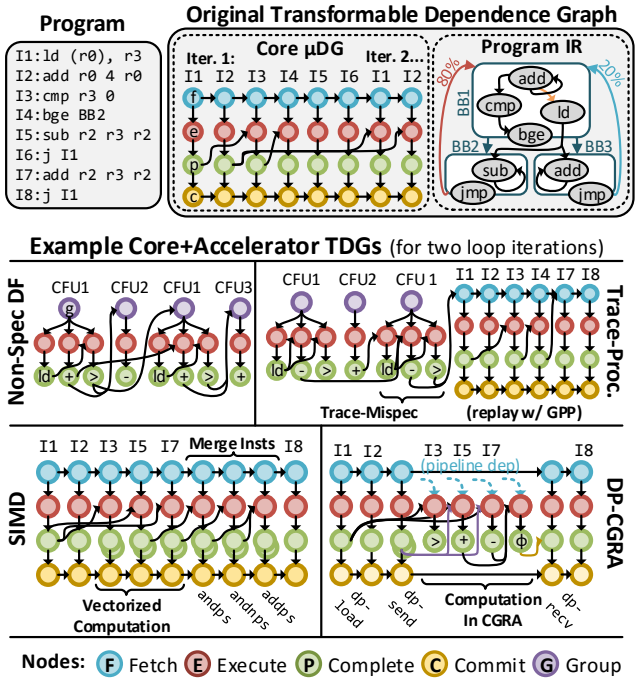

Figure 8: Example TDGs

between loop iterations can be detected by tracking per-iteration memory addresses in consecutive iterations. Loops with non-vectorizable memory dependences are excluded, and considering loop-splitting and loop-reordering to break these dependences is future work. Similarly, loops with inter-iteration data dependences which are not reductions or inductions are excluded.

For vectorizing control flow, the TDG analysis considers an if-conversion transformation, where basic blocks in an inner-loop are arranged in reverse-post order, and conditional branches become predicate-setting instructions. In the case of non-local memory access, packing/unpacking instructions are inserted into the loop body. For alignment, we assume we have the use of unaligned memory operations (like in x86). This analysis also computes where masking instructions would need to be added along merging control paths. The TDG decides whether to vectorize a loop by computing the expected number of dynamic instructions per iteration by considering path profiling information. If it is more than twice the original, the loop is disregarded.

TDG Transform ($TDG_{GPP,\emptyset}$ to $TDG_{GPP,SIMD}$): When a vectorizable loop is encountered, μDG nodes from the loop are buffered until the vector-length number of iterations are accumulated. The first iteration of this group becomes the *vectorized* version, and not-taken control path instructions, as well as mask and predicate instructions, are inserted. Most instructions are converted to their vectorized version, except for non-contiguous loads/stores, for which additional scalar operations are added (as we target non-scatter/gather hardware). At this point, memory latency information is remapped onto the *vectorized* iteration, and the non-vector iterations are elided. If fewer than the minimum vector length iterations remain, the SIMD transform is not used.

Data-Parallel CGRA (DP-CGRA) TDG *TDG Analysis:* The analysis "plan" is a set of legal and profitable loops, potentially vectorized, where for each loop the plan contains the *computation subgraph* (offloaded instructions). Vectorization is borrowed from SIMD, and a slicing algorithm [17] is used to separate instructions between the general core and CGRA. Control instructions without forward memory dependences are offloaded to the CGRA.

Similar to SIMD, a new version of the inner loops is constructed, except here the computation subgraph is removed from the loop, and communication instructions are inserted along the interface edges. If the loop is vectorizable, the computation can be "cloned" until its size fills the available resources, or until the maximum vector length is reached, enabling more parallelism. The analysis algorithm disregards loops with more communication instructions than offloaded computation.

TDG Transform ($TDG_{GPP,\emptyset}$ to $TDG_{GPP,DP\text{-}CGRA}$): The DP-CGRA keeps a small configuration cache, so if a configuration is not found when entering a targeted loop, instructions for configuration are inserted into the TDG. Similar to SIMD, μDG nodes from several loop iterations are buffered until the vectorizable loop length is reached. At this point, if the loop is vectorizable, the first step is to apply the SIMD transformation as described earlier ($TDG_{GPP,\emptyset}$ to $TDG_{GPP,SIMD}$).

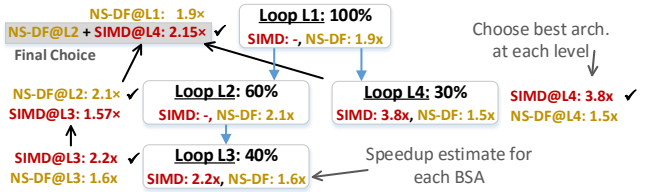

Figure 9: Amdahl Tree – Example of Triple Nested Loop

Offloaded instructions require two additional edges to enforce accelerator pipelining: one for the pipeline depth between computation instances, and one for in-order completion. We model the scheduling and routing latency by adding delay on the data dependence edges.

Non-speculative Dataflow (NS-DF) TDG *TDG Analysis:* The primary analysis here is to find fully inlinable loops or nested loops that fit within the hardware budget. Once a loop nest selected, the control is converted into data-dependences by computing the program dependence graph. This determines where "switch" instructions should be inserted to handle control. Instructions are then scheduled onto CFUs using known mathematical optimization [36] techniques.

TDG Transform ($TDG_{GPP,\emptyset}$ to $TDG_{GPP,NS\text{-}DF}$): This transform operates at basic-block granularity, by inserting edges to enforce control dependences and to force each compound instruction to wait for all operands before beginning execution. It also adds edges to enforce writeback network capacity. Additionally, edges between the general core and NS-DF regions are inserted to model live value transfer time.

Trace-Processor (Trace-P) TDG *TDG Analysis:* The analysis plan is a set of eligible and profitable inner loops and compound instruction schedules. Eligible loops with hot traces are found using path profiling techniques [4]. Loops are considered if their loop back probability is higher than 80%, and their configuration size fits in the hardware limit. A similar CFU scheduling technique is used, but compound instructions are allowed to cross control boundaries.

TDG Transform ($TDG_{GPP\text{-}Orig,\emptyset}$ to $TDG_{OOO,Trace\text{-}P}$): This transform is similar to NS-DF, except that the control dependences are not enforced, and if Trace-P mispeculates the path, instructions are replayed on the host processor by reverting to the $TDG_{GPP\text{-}Orig,\emptyset}$ to $TDG_{GPP\text{-}New,\emptyset}$ transform.

3.3 BSA Selection

A practical consideration is how to choose between BSAs throughout the program execution. This is complicated by the fact that the decision is hierarchical in program scope (eg. target an entire loop nest, or just the inner loop?).

We propose a simple strategy called the Amdahl Tree, as shown in Figure 9. Each node in the tree represents a candidate loop, and is labeled with the speedup of each BSA, and the expected execution time. This speedup can be approximate based on static or profile information. A bottom-up traversal, applying Amdahl's law at each node, can determine the best architecture for a given region. In practice, a profile-based compiler can make BSA selections and embed them in the program binary.

Suite	Benchmarks
TPT	conv, merge, nbody, radar, treesearch, vr
Parboil	cutcp, fft, kmeans, lbm, mm, needle, nnw, spmv, stencil, tpacf
SPECfp	433.milc 444.namd 450.soplex 453.povray 482.sphinx3
Mediabench	cjpeg, djpeg, gsmdecode, gsmencode cjpeg2, djpeg2, h263enc, h264dec, jpg2000dec, jpg2000enc, mpeg2dec, mpeg2enc
TPCH	Queries 1 and 2
SPECint	164.gzip, 181.mcf, 175.vpr, 197.parser, 256.bzip2 429.mcf, 403.gcc, 458.sjeng, 473.astar, 456.hmmer, 445.gobmk

Table 3: Benchmarks

	IO2	OOO2	OOO4	OOO6
Fetch, Dispatch Issue, WB Width	2	2	4	6
ROB Size	-	64	168	192
Instr. Window	-	32	48	52
DCache Ports	1	1	2	3
FUs (ALU,Mul/Div,FP)	2,1,1	2,1,1,	3,2,2	4,2,3

Table 4: General Core Configurations

4. ExoCore Exploration Methodology

The following methodology is used in the design-space exploration in the next section.

Benchmarks Selection Benchmarks were chosen from a wide range of suites (Table 3). These include highly regular codes from Intel TPT [17], scientific workloads from PARBOIL [1], image/video applications from Mediabench [27] and irregular workloads from SPECint. The diversity highlights ExoCore's ability to target a large variety of codes. Also, as much as possible, we picked benchmarks and suites from the respective accelerator publications.

General Core Configurations We considered four different cores of varying complexity, with parameters as outlined in Table 4. The common characteristics are a 2-way 32KiB I$ and 64KiB L1D$, both with 4 cycle latencies, and a 8-way 2MB L2$ with a 22 cycle hit latency. We model 256-bit SIMD.

Area Estimation We use McPAT for estimating area of general cores, and use area estimates from relevant publications [17, 18, 36].

Runtime Accelerator Selection Because we are exploring the potential of ExoCore systems, most of our results use an Oracle scheduler, which chooses the best accelerator for each static region, based on past execution characteristics. The selection metric we use is energy-delay, where no individual region should reduce the performance by more than 10%. One later set of results compares the oracle and Amdahl Tree schedulers.

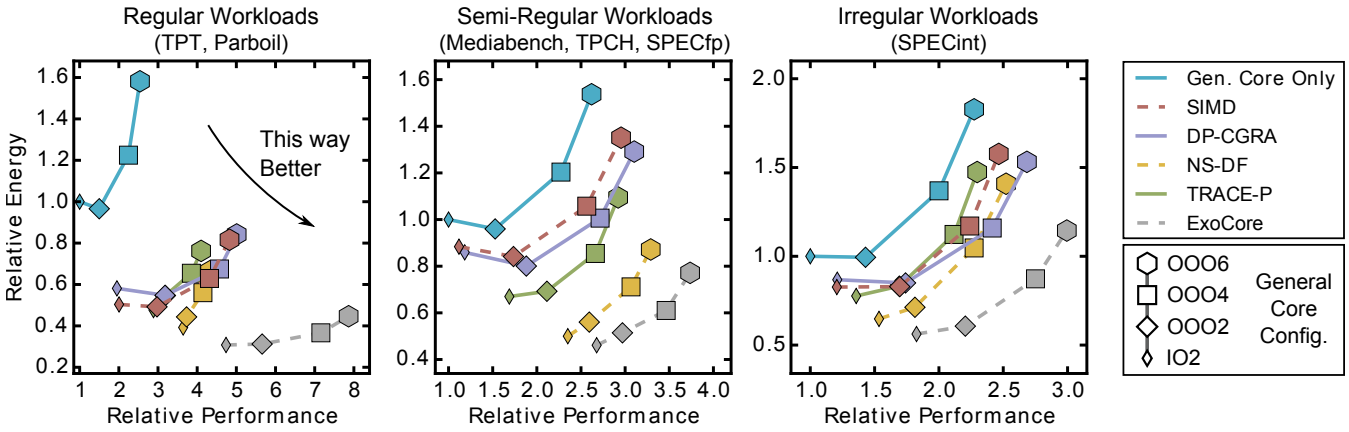

Figure 11: Interaction between Accelerator, General Core, and Workloads

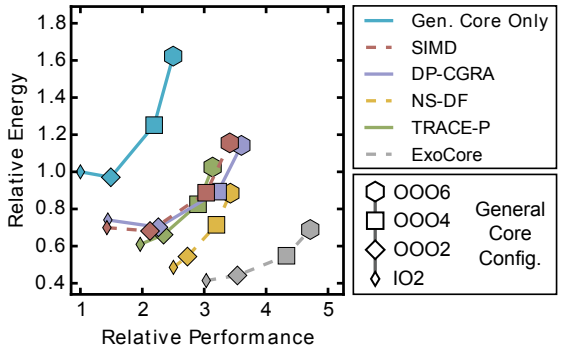

Figure 10: ExoCore Tradeoffs Across All Workloads

5. Design Space Exploration

In this section, we explore a design space comprising the combinations of four general purpose cores and 16 choices for the subset of BSAs (from the set of four BSAs we study). For this section, a "full ExoCore" consists of an ExoCore with all four BSAs. We report three important results first:

- **High Potential of BSAs** Across all workloads, a full OOO2-based ExoCore provides 2.4× performance and energy benefits over an OOO2 core (dual-issue, out-of-order). Compared to an OOO6 core, an OOO6 ExoCore can achieve up to 1.9× performance and 2.4× energy benefits.
- **Design Choice Opportunities** Our analysis suggests that there are several ways to break through current energy, area, and performance tradeoffs. For example, there are four OOO2-based ExoCores and nine OOO4-based ExoCores that achieve higher performance than an OOO6 with SIMD alone. Of these, the $OOO2_{SIMD+DP-CGRA+NSDF}$ ExoCore achieves 2.6× better energy efficiency, while requiring 40% less area.
- **Affinity Behavior** ExoCores make use of multiple accelerators, both inside workload domains, and inside applications themselves. Across all benchmarks, considering a full OOO2 ExoCore, an average of only 16% of the original programs' execution cycles went un-accelerated.

We discuss each of these results in the following subsections. We also again emphasize that, in principle, the same analysis could be done by implementing and integrating many compilers and simulators inside the same framework. However it is difficult and impractically time-consuming. See Appendix A for steps in TDG modeling.

5.1 High Potentials of ExoCores

Overall Trends Figure 10 shows geometric mean performance and energy benefits of single-BSA designs as well as full ExoCores across all workloads. Each line in this graph represents the set of designs with the same combination of accelerators (or no accelerators), and each point on the curve represents a different general purpose core. As alluded to, we show that each BSA alone has significant potential, and their combination has even more.

Workload Interaction To observe workload domain specific behavior, Figure 11 divides the previous results into highly regular, semi-regular, and highly irregular workload categories. The major result here is that even on the most challenging irregular SPECint applications, ExoCores have significant potential. A full OOO2 ExoCore can achieve 1.6× performance and energy benefits over OOO2 with SIMD. For OOO6, ExoCore achieves 1.25× performance and 50% energy efficiency improvement. Our findings also confirm the intuitive result that BSAs have a high potential on regular workloads, where ExoCore achieves 3.5× performance and 3× energy improvement for OOO2 core.

5.2 Design Choice Opportunities

An ExoCore organization opens design opportunities which can push traditional energy and performance tradeoffs forward, and this is true for both complex and simple general purpose cores. The TDG enables the exploration of these designs.

To demonstrate this capability, and examine some new opportunities, we perform a design space exploration across

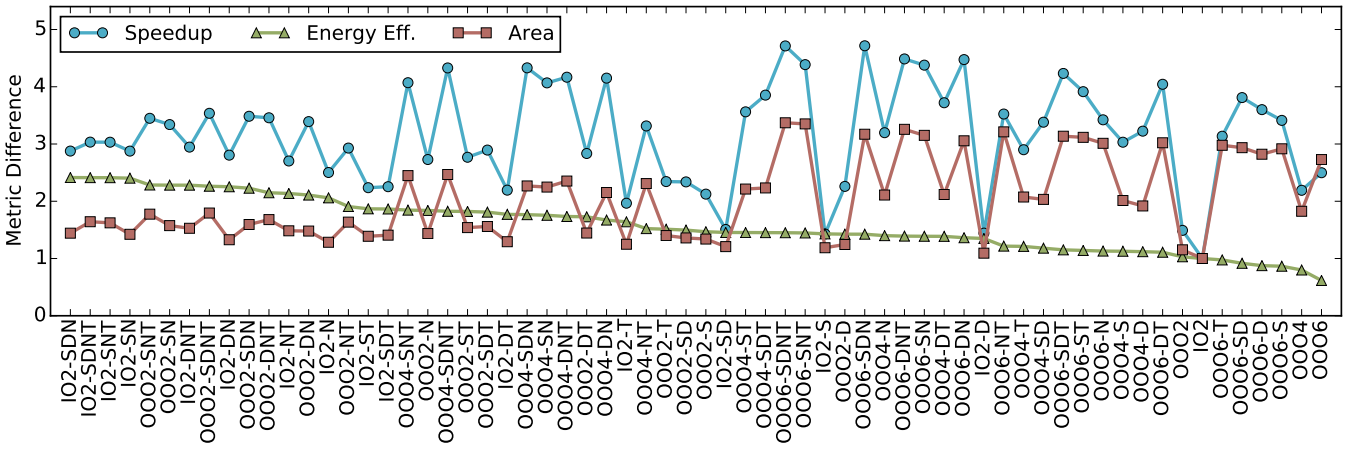

Figure 12: Design-Space Characterization. S: SIMD, D: Data-Parallel CGRA, N: Non-spec Dataflow, T: Trace-Processor

all combinations of general purpose cores and BSAs across all workloads. The resulting performance, energy efficiency and area metrics are shown in Figure 12, where all points are relative to the dual-issue in-order (IO2) design. On the graph, higher is better for performance and energy efficiency, and lower is better for area. There are many valuable quantitative insights.

- **[Performance]** Compared to the OOO6 core with SIMD[5], four OOO2 and nine OOO4 ExoCore configurations match the performance with lower energy and area, by as much as 2×. This gives the designer freedom even if performance is a hard constraint.
- **[Performance]** No in-order ExoCore configuration can match the OOO6 performance. The best achieved is 88% of the OOO6, with almost 1.7× lower area.
- **[Energy]** The OOO2 core with SIMD is the most energy efficient baseline core. Compared to this core, **twelve** in-order and **five** OOO4 ExoCores have higher energy efficiency - by as much as 65% and 25% respectively. In addition to being more energy efficient, these configurations are up to 1.35× (IO2-SDN) and 1.9× (OOO4-SNT) higher performance.
- **[Energy]** A full OOO6 ExoCore achieves the same energy efficiency (within 1%) of the OOO2-SIMD core and has 2.2× higher performance, but is nearly twice the area.
- **[Full ExoCores]** The full IO2 ExoCore is the most energy-efficient among all designs. The full OOO6 ExoCore has the best performance, the next best is the full OOO4 ExoCore, which has 10% lower performance, 1.25× lower energy and 1.36× lower area.

Overall these results suggest two trends. Behavior specialized cores are extremely effective at reducing energy regardless of core type. They provide high performance-improvements for in-order, small and medium-sized OOO cores - but not much for big OOO cores. Therefore, the most high impact opportunity for future BSAs is to improve the performance of OOO cores, while further increasing their energy efficiency. The TDG enables this and other such explorations.

[5] We use OOO6-SIMD as the baseline - since this resembles commercial processors with AVX2 etc. Note that if we had used OOO6 without SIMD as the baseline, the benefits of ExoCores would be even higher.

5.3 Affinity Behavior

We have thus far considered aggregate metrics across workloads, but the TDG also enables application-level analysis, which we perform next.

Inter-application Affinity Figure 13 considers an OOO2-based ExoCore, and shows each benchmark's energy and execution time breakdown. The baseline is the OOO2 alone. Most benchmarks, even irregular workloads, have some potential to effectively use BSAs, ranging from 100% of the program being offloaded, to around 20%. Secondly, some workloads can benefit from multiple BSAs inside a single application. For example, cjpeg-2 makes use of SIMD, NS-DF and Trace-P during its execution.

Considering energy breakdowns in Figure 13, observe that for SIMD and DP-CGRA, the percentage of cycles offloaded is proportional to the energy contribution. For NS-DF, because it can power-gate portions of the general core when in use, the energy reduction is higher than the execution time reduction.

Affinity Granularity To give insight into the dynamic behavior of an ExoCore, Figure 14 shows a trace of execution for two benchmarks. These graphs show the relative performance benefits of a full OOO2 ExoCore over the OOO2 core alone, over time. These graphs demonstrate both that applications can have fine-grain affinity for different accelerators, and that a TDG-based framework can capture and model this behavior.

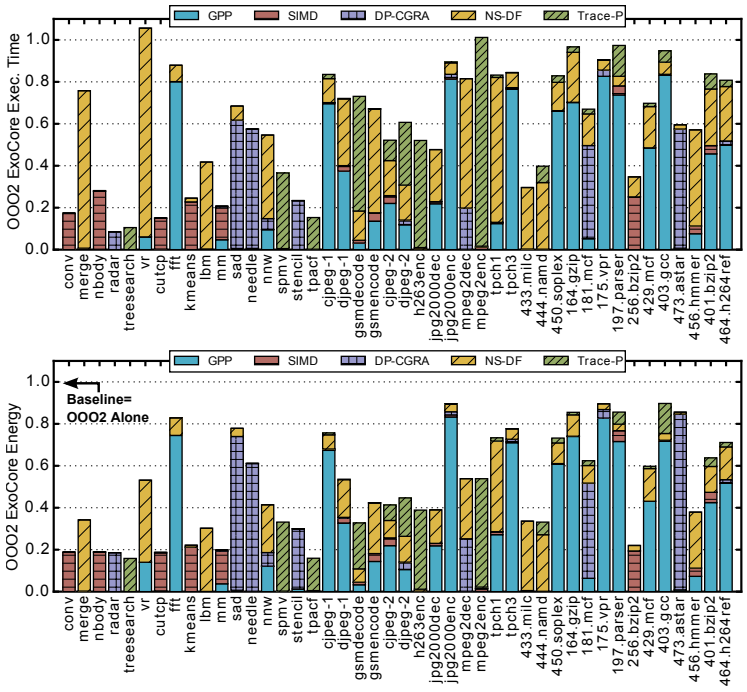

Figure 13: Per-Benchmark Behavior and Region Affinity

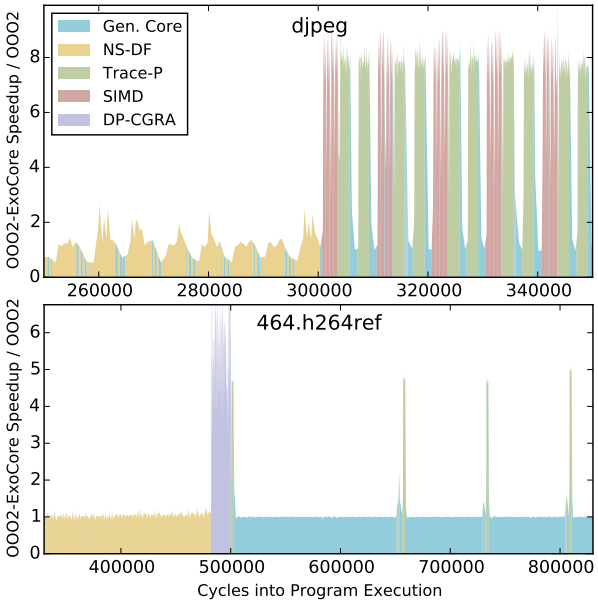

Figure 14: ExoCore's Dynamic Switching Behavior

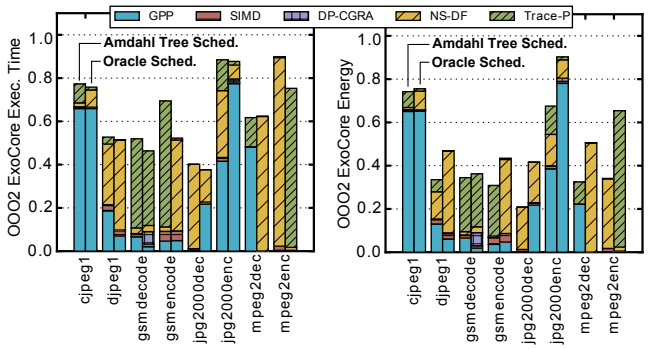

Figure 15: Oracle versus Amdahl Tree Scheduler

5.4 Practicality

Though we have shown that ExoCores have significant potential, an important concern is whether they can practically provide benefits without oracle information. To study this, Figure 15 presents a comparison of the performance and energy of the Amdahl scheduler (left-bar) and the Oracle scheduler (right-bar). Here, the general core is the OOO2, and we are showing challenging benchmarks from Mediabench which require using multiple different accelerators in the same application to be effective.

Compared to the oracle scheduler, our scheduler is slightly over-calibrated towards using the BSAs rather than the general core, meaning it is biased towards energy efficiency – across all benchmarks (including those not shown) the scheduler provides 1.21× geomean energy efficiency improvement, while giving 0.89× performance of the Oracle scheduler. These heuristics can be tuned with more effort to favor different metrics, or to be useful for the OOO4 or OOO6 cores.

5.5 Evaluation Limitations and Extensions

This evaluation has explored *a* design space across cores, accelerators and various metrics. Of course, there is a much larger design space including varying core and accelerator parameters, modifying their frequencies, and including other proposed or novel accelerators. Though we have admittedly just scratched the surface, the TDG can enable insightful modeling of this design space.

The TDG framework and the benefits of combining BSAs suggests that a fruitful area of future work is to see if an ExoCore can be designed to match domain specific accelerators in certain domains.

Also, our TDG transforms are simply written as short functions in C/C++. A DSL to specify these transforms could make the TDG framework even more productive for designers.

6. Related Work

Parallel and Heterogeneous Architecture Modeling An alternative to modeling architectures with the TDG are analytical models. Relevant works include those which reason about the benefits of specialization and heterogeneity [11, 20, 45, 57], high-level technology trend projections [6, 22], or even general purpose processors [10, 12, 14]. There are also several analytical GPU models [21, 32, 39, 40, 46, 54].

The drawback of such models is that they are either specific to the modeling of *one* accelerator or general-purpose core, or they are too generic and do not allow design space explorations which capture detailed phenomenon.

Kismet [16, 23] is a model which uses profiles of serial programs to predict upper-bound speedups for parallelization. It uses a hierarchical critical path analysis to characterize the available and expressible parallelism.

Perhaps the most related technique is the Aladdin framework [44], a trace-based tool that uses a compiler IR interpreter, that enables design space exploration of *domain specific accelerators*. Using such an approach for behavioral specialized accelerators is possible, and should reduce errors from ISA artifacts. The drawback would be that the interaction with the general purpose core in that framework would be more difficult to capture. However, that style of approach may be an avenue for improving accuracy in the future.

Core Heterogeneity Single-ISA heterogeneous architectures have been extensively explored, starting with Kumar et al. [25]. Later work extended this by exploring a large design space of microarchitectural design points [26], and Lee et al. use regression analysis to broaden this design space even further. A related design approach is to use a shared pipeline frontend, but use a heterogeneous backend inside the microarchitecture, like Composite Cores [31].

To a lesser extent, multi-ISA heterogeneity (where all cores are general purpose) have been previously studied, including Venkat et al. [52], who show that heterogeneity in the ISA alone can provide benefits. Our work considers the composition of accelerator-type architectures, which offer tradeoffs beyond those of general purpose architectures.

Relatedly, many past works have explored processor steering methods on heterogeneous architectures [35, 41, 48, 51, 55]. The unique aspect of the scheduling problem in this work is that entry points to different accelerators are restricted by the program's loop structure.

7. Conclusions

In this work, we proposed the ExoCore design which incorporates multiple behavior-specialized accelerators inside a core, and a comprehensive and novel modeling methodology for studying such architectures called the transformable dependence graph (TDG).

The TDG consists of a closely-coupled μDG and Program IR for analysis. This representation allows the study of the combined effects of compiler and hardware microarchitecture as graph transformations. We showed the TDG approach is accurate and allows deep and insightful analysis and design-space exploration.

Broadly, the ExoCore approach could be influential in two ways. First, it simplifies general purpose core design – behavior-specific microarchitecture blocks can be designed and integrated into the core in a modular fashion, without disruptive changes to the core's microarchitecture. Second, it provides a promising approach for exceeding the performance/energy frontier of conventional approaches. Finally, an open question that arises from this paradigm is how powerful must the baseline core be - can one design sufficiently powerful and expressive BSAs, where only a simple in-order core is sufficient, and most of the application time is spent in a BSA.

A. Steps in TDG Model Construction

Here we discuss the practical aspects and steps in constructing a TDG model.

Analysis The first step is identifying the required compiler analysis or profiling information, and implementing a pass to compute it, operating on the IR or trace respectively. Often, the required information (eg. path profiling) already exists because it is common among BSAs. When this information cannot be computed, approximation may be appropriate.

Transformations The next step is to write an algorithm (a "transform") which reads the incoming μDG trace, and modifies dependences to model the behavior of the BSA in question. Depending on the granularity of acceleration, a transform may operate on a single instruction at a time, or it might collect a basic block, loop iteration, or several iterations before it can decide what the final μDG should be. The modified μDG is transient, and is discarded after any analysis (eg. critical path analysis), once it is no longer inside the current instruction window. A transformation should also include a model for how it interacts with the core when it enters and exists a region.

Scheduling Finally, the model must decide *when* to apply the BSA transform (ie. at what point in the code). In a single-BSA system, the BSA's transform can be used at any legal entry point. For multi-BSA systems, the model should provide per-region performance (or other metric) estimates relative to the general purpose core for the BSA based on either the IR or profiling information. This is used with the Amdahl tree to decide which BSA to use in each region.

Validating new BSAs Validating a TDG model of newly-proposed BSAs is similar to validating a cycle-level simulator. Writing microbenchmarks and sanity checking is recommended (eg. examining which edges are on the critical path for some code region).

Acknowledgments

We thank Venkatraman Govindaraju for his help in creating the initial TDG models and validation for DySER and SIMD. We also thank our shepherd, David Wentzlaff, and the anonymous reviewers for their comments and advice.

Support for this research was provided by NSF under the grants CNS-1218432 and CCF-0917238, and by a Google PhD Fellowship.

References

[1] *Parboil Benchmark Suite*. impact.crhc.illinois.edu/parboil/parboil.aspx.

[2] *Vertical Microbenchmarks*. http://cs.wisc.edu/vertical/microbench.

[3] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. In *ISCA*, 2014.

[4] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO*, 1996.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

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

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[8] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO*, 2004.

[9] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA*, 2001.

[10] Lieven Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 2010.

[11] Peter A. Milder Eric S. Chung, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPUs? In *MICRO '10*.

[12] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 2011.

[13] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[14] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 2009.

[15] Brian Fields, Shai Rubin, and Rastislav Bodik. Focusing processor policies via critical-path prediction. In *ISCA*, 2001.

[16] Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *PLDI*, 2011.

[17] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy efficient computing. *IEEE Micro*, 2012.

[18] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO*, 2011.

[19] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA '10*.

[20] Mark Hempstead, Gu-Yeon Wei, and David Brooks. Navigo: An early-stage model to study power-contrained architectures and specialization. In *Proceedings of Workshop on Modeling, Benchmarking, and Simulations (MoBS)*, 2009.

[21] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. In *ISCA '10*.

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

[23] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: Parallel Speedup Estimates for Serial Programs. In *OOPSLA*, 2011.

[24] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *MICRO*, 2013.

[25] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO*, 2003.

[26] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT*, 2006.

[27] Chunho Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, 1997.

[28] Jaewon Lee, Hanhwi Jang, and Jangwoo Kim. Rpstacks: Fast and accurate processor design space exploration using representative stall-event stacks. In *MICRO*, 2014.

[29] Sheng Li, Jung-Ho 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*, 2009.

[30] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. PuDianNao: a polyvalent machine learning accelerator. In *ASPLOS*, 2015.

[31] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite Cores: Pushing heterogeneity into a core. In *MICRO*, 2012.

[32] J. Meng, V.A. Morozov, K. Kumaran, V. Vishwanath, and T.D. Uram. GROPHECY: GPU performance projection from CPU code skeletons. In *SC'11*. ACM, 2011.

[33] Tipp Moseley, Dirk Grunwald, Daniel A Connors, Ram Ramanujam, Vasanth Tovinkere, and Ramesh Peri. Loopprof:

Dynamic techniques for loop detection and profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[34] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 2009.

[35] Sandeep Navada, Niket K. Choudhary, Salil V. Wadhavkar, and Eric Rotenberg. A unified view of non-monotonic core selection and application steering in heterogeneous chip multiprocessors. In *PACT*, 2013.

[36] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous Von Neumann/Dataflow execution models. In *ISCA*, 2015.

[37] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Pushing the limits of accelerator efficiency while retaining programmability. In *HPCA*, 2016.

[38] Tony Nowatzki, Venkatraman. Govindaraju, and Karthikeyan Sankaralingam. A graph-based program representation for analyzing hardware specialization approaches. *Computer Architecture Letters*, 2015.

[39] Cedric Nugteren and Henk Corporaal. A modular and parameterisable classification of algorithms. Technical Report ESR-2011-02, Eindhoven University of Technology, 2011.

[40] Cedric Nugteren and Henk Corporaal. The boat hull model: adapting the roofline model to enable performance prediction for parallel computing. In *PPOPP*, 2012.

[41] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Trace based phase prediction for tightly-coupled heterogeneous cores. In *MICRO*, 2013.

[42] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution engine: Balancing efficiency and flexibility in specialized computing. In *ISCA*, 2013.

[43] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *MICRO*, 1997.

[44] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ISCA*, 2014.

[45] M. Shoaib Bin Altaf and D.A. Wood. LogCA: a performance model for hardware accelerators. *Computer Architecture Letters*, 2015.

[46] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *PPoPP*, 2012.

[47] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, 2000.

[48] Tyler Sondag and Hridesh Rajan. Phase-based tuning for better utilization of performance-asymmetric multicore processors. In *CGO*, 2011.

[49] Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. Architectural specialization for inter-iteration loop dependence patterns. In *MICRO*, 2014.

[50] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *MICRO*, pages 291–, 2003.

[51] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *ISCA*, 2012.

[52] Ashish Venkat and Dean M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *ISCA*, 2014.

[53] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *ASPLOS '10*.

[54] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 2009.

[55] Jonathan A. Winter, David H. Albonesi, and Christine A. Shoemaker. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *PACT*, 2010.

[56] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. In *ASPLOS*, 2014.

[57] T. Zidenberg, I. Keslassy, and U. Weiser. Optimal resource allocation with multiamdahl. *Computer*, 2013.