Studying Hybrid Von-Neumann/Dataflow Execution Models

A Graph-Based Program Representation For Analyzing Hardware Specialization Approaches

Tony Nowatzki  Venkatraman Govindaraju  Karthikeyan Sankaralingam
University of Wisconsin - Madison

Abstract

Hardware specialization is becoming a promising paradigm for future microprocessors. Unfortunately, by its very nature, the exploration of specialization ideas (the artifact is dubbed an “accelerator”) are developed, evaluated, and published as end-to-end vertical silos spanning application, language/compiler, and hardware architecture, with per-accelerator customized tools, and little opportunity for cross-application of ideas from one accelerator into another.

This paper develops a novel program representation suitable for the hardware specialization paradigm, called the transformable dependence graph (TDG), which combines semantic information about program properties and low-level hardware events (cache misses, branch mis-predictions, resource hazards, energy expended by hardware events) in a single representation.

We demonstrate that the TDG is a feasible, simple, and accurate modeling technique for transparent specialization approaches, enabling architectures to be compared and analyzed easily in a single framework. We demonstrate models for four previously proposed accelerators.

1. Introduction

Due to diminishing returns from technology scaling, computer architects have turned to “specialization” as one paradigm for providing successive generations of improved performance and lower power and energy. Broadly, this paradigm of specialization is straight-forward: pick a problem “domain,” identify the primitives, and build hardware abstractions for these primitives. In highly simplified terms, a primitive could mean an operation (like sigmoid operations in neural processing or convolution in image processing), or a data access mechanism (like random accesses not amenable to caching). The insight of this paradigm is that general purpose processing at instruction granularity is too inefficient compared to the specialization alternative of implementing and organizing these primitives directly in hardware.

Within this broad specialization paradigm there are two philosophical approaches - exposed specialization and transparent specialization. In the exposed approach, the compiler and conventional programming is largely eschewed, and an accelerator is built for a specific domain (or algorithm), and exposed to the programmer via a programming library.

The alternative paradigm, which is the focus of this paper is transparent specialization. Here the programmer continues to program their applications and algorithms of interests in established languages like C/C++/Java/Python etc. The insight is to identify program behavior and their inter-relationship to hardware microarchitecture and develop mechanisms that are more efficient than today’s processors. The approach is generally domain agnostic and typically is integrated into conventional microprocessors sharing the memory hierarchy (caches etc.).

The most widespread and established commercial example is SIMD or short-vector specialization where “regular” computation and memory access can be exploited by expressing, in a single instruction, operations on multiple pieces of data.

This paper explores the question of modeling transparent specialization, including the interaction and composition of multiple accelerators, compiler design, design of future specialization techniques, and interaction with different general purpose processor (GPP) designs.

Our Proposal This work proposes a novel program representation called the transformable dependence graph (TDG) which combines semantic information about program properties and low-level hardware events (cache misses, branch mis-predictions, resource hazards, energy expended by hardware events) in a single representation (note this is a representation of the dynamic execution of the program - not a static representation like a DFG/CFG/PDG). An overview of the approach is in Figure 1, described in detail in the next section.

This representation captures program transformations and compiler optimizations like vectorization, trace-optimization, and if-conversion and acceleration hardware specialization techniques like compound functional units and coarse-grained reconfigurable computing. These can be expressed as graph transformations of this program representation, allowing one to obtain the execution time, power, and energy of executing a program on an accelerator.

Findings and Results Our results demonstrate the following: The TDG framework is feasible, practical, and accurate. It allows us to accurately represent four diverse transparent specialization techniques using a single framework (which have thus far been studied with per-accelerator compilers, sim-
We first explain why existing representations are insufficient, in- 
manance and do not allow design space explorations. Recently, a 
one are specific to 
analytical models for general purpose processors [9, 6]. These 
purpose processors [8]. Models for heterogeneous execution 
cent papers which have presented a case for specialization and 
idea of developing a rich program representation, are many re-

Other related approaches Orthogonal, but related to the 
other the idea of developing a rich program representation, are many re-
cent papers which have presented a case for specialization and heterogeneity using analytical models [7, 31, 16], high-level 
technology trend projections [5, 18], or by modeling general 
purpose processors [8]. Models for heterogeneous execution 
include [25, 24, 30, 17, 27, 21, 29, 22], which are related to analytical models for general purpose processors [9, 6]. These 
are specific to one accelerator or general-purpose core perfor-
mance and do not allow design space explorations. Recently, a 
framework called Aladdin allows design space exploration of 
exposed specialization, but has not yet demonstrated capability 
of modeling transparent specialization [26].

2.2. The Transformable Dependence Graph

Leveraging Microarchitectural Dependence Graphs (µDGs) To satisfy the first requirement, we 
turn to the µDG, which is a higher-level representation of microarchitectural execution, composed of nodes for 
microarchitectural events and edges for their dependences. It is 
constructed using dynamic information from a simulator, and 
has traditionally been used for modeling GPP cores [12]. 
The µDG offers the correct level of detail for modeling 
microarchitecture, yet is still abstract and easy to model 
modifications/additions of effects.

The second requirement – being able to capture com-
piler/application interactions – can be accomplished by re-
constructing and analyzing a program IR with a direct one-to-one 
mapping with dynamic instructions.

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 1, a simulator produces dynamic instruc-
tion, dependence, and microarchitectural information, which 
is used by the constructor to build the TDG. The TDG is ana-
alyzed 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 GPP core and accelerator. As part of 
the representation, the TDG carries information about overall 
execution time and energy. The next subsection describes the 
components of the approach in detail.

Note that we are the first to demonstrate that energy model-
ing can be performed using µDGs, and we are also the first to 
recognize that the end-effect of acceleration can be represented 
by performing graph modifications to the µDG.

Notation To aid exposition, the notation TDGGPP,ACCEL 
refers to a TDG representation of a particular GPP and accel-
erator. For example, TDGOOO4SIMD represents a quad-issue 
OOO GPP with a SIMD accelerator. As a special case, the 
original TDG prior to any transformations (not representing 
an accelerator) is TDGGPP,∅.

2.3. Defining the Transformable Dependence Graph

Here we define the components of our approach using a run-
ing example in Figure 2, which is for transparently applying 
a simple fused multiply-accumulate (fma) instruction.

Constructing the TDG To construct TDGGPP,∅, a conven-
tional OOO GPP simulator (like gem5 [4]) executes an 
modified binary, and feeds dynamic information to the TDG 
constructor (Figure 2(a)). The first responsibility of the tool is 
to create the original µDG, which embeds dynamic microarchi-
tectural information including data and memory dependences, energy events, dynamic memory latencies, branch mispredicts and memory addresses.

To explain, inside Figure 2(b) is an example 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 ($D_{1-2} \rightarrow D_{1-2}, C_{1-2} \rightarrow C_{1-2}$). The FU or memory latency is represented by edges from execute to complete ($E_{1} \rightarrow P_{1}$), and data dependencies by edges between complete to execute ($P_{1} \rightarrow E_{1}$). Note that the μDG represents the dynamic execution of the program, and can be large; for performance reasons we use a windowed approach described in Section 4.

The second responsibility of the constructor is to create a program IR (also in Figure 2(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 [23]. Also, register spill and constant access is identified for later optimization. To account for not-taken control paths in the program, we augment the program IR with the static paths from straightforward binary analysis.

**TDG Analyzer** The next step is to analyze the TDG to determine which program regions can be 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 2(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.

**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 TDDG$_{GPP,Y}$ to TDDG$_{GPP,Y,ACCEL}$.

Figure 2(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 to the associated fma.

### Core+Accelerator TDG

The core+accelerator TDG represents their combined execution, and an example of which is in Figure 2(e), for TDDG$_{GPP,Y,fma}$. Here, 12$'$ represents the specialized instruction, and 13 has been eliminated from the graph. In practice, accelerators can be more coarse grain and modify the TDG in deeper ways (examples in Section 3).

Finally, this TDG can be analyzed for performance and power consumption. The length of the critical path, shown in bold in the figure, determines the execution time in cycles. For energy/power, we associate events with nodes and edges, which can be accumulated and fed to standard energy modeling tools. The implementation of our framework is discussed in Section 4.

#### Limitations

As we have so far only encountered transparent accelerators which attach to the GPP cache system, we have not attempted to model (or express) the effects of switching between memory systems, and this is future work. GPUs, another popular and established accelerator, is programmable, but often requires algorithm changes to the CPU version of a program, and hence we classify it as an exposed accelerator and leave it outside the scope of TDG.

---

1In practical terms, if one had GPU code, it is a logistic matter of building a TDG constructor that reads GPU assembly and interprets GPU simulator events. The rest of our approach can be re-used. Alternatively, GPU analytical models [17] can be combined with TDG analysis.

---

**Table 1: Prism’s Intra-Instruction Nodes and Edges**

<table>
<thead>
<tr>
<th>Edge Constraint</th>
<th>Latency</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F \rightarrow D$</td>
<td>Fixed</td>
<td>✓</td>
</tr>
<tr>
<td>$D \rightarrow E$</td>
<td>Dispatch before Exec. 0</td>
<td>✓</td>
</tr>
<tr>
<td>$E \rightarrow G$</td>
<td>Group before Exec. 0</td>
<td>✓</td>
</tr>
<tr>
<td>$E \rightarrow P$</td>
<td>Execution Latency</td>
<td>Fixed</td>
</tr>
<tr>
<td>$P \rightarrow C$</td>
<td>Load Latency</td>
<td>Recorded</td>
</tr>
<tr>
<td>$C \rightarrow W$</td>
<td>Store Latency</td>
<td>Recorded</td>
</tr>
</tbody>
</table>
3. TDG Transform Implementations

This section describes the TDG analysis and transformation algorithms for the GPPs and accelerators, and we begin by describing some preliminaries to ease explanations.

Subsequently, in describing transformations, we assume that TDG GPP-Orig, has already been constructed, and we are transforming to TDG GPP-New, ACCEL. Though we describe GPP and accelerator transformations separately, these would be applied in-concert to create a new TDG. Figure 3 gives example TDGs for all architectures on a simple program.

Also, in our presentation, we avoid formal algorithmic specification, and instead describe models in informal terms to aid in exposition. One of the contributions of this work is that these are straightforward transformations. Future work will explore formal graph-theory representations of transformations.

3.1. Preliminaries

In a TDG, the behavior of each dynamic instruction or operation is represented by a set of nodes for each of its stages of execution. Edges represent dependences between these stages, both inside and across instructions. Table 1 shows the nodes and inter-instruction edges and their latencies for all GPP and accelerator models discussed in this paper, and Table 2 describes inter-instruction edges.

Essential Components To give some intuition of the purpose of various nodes and edges, we describe common components here, specifically the Execute (E), Complete (P) and Writeback (W) nodes. The $E \rightarrow P$ edge captures FU and load latency, while the edge into W ($C \rightarrow W/P \rightarrow W$) captures store latency. The $P \rightarrow E$ edge captures data and memory dependences between instructions. The GPP model also includes Fetch (F) Dispatch (D) and Writeback (W) nodes for representing pipeline behavior. Other architecture-specific nodes and edges will be described as-needed.

Representing Resources with Dynamic Edges It is simple to represent microarchitectural resources which are acquired and released in-order, like the one which represents the ROB size ($C_{\text{robsize}} \rightarrow D$). However, many resources in an architecture are not acquired and released in a pre-specified order (e.g. functional units).

Our approach for representing these resources is to keep a cycle-indexed data-structure that tracks which resources are occupied by which originating instruction and when they will be freed. If all of a certain resource are taken, a dependence is added between the first node which frees the given resource to the node which is requesting the resource. We refer to these edges as dynamic edges, because their source and destination instructions change depending on the timing of other events. These edges are indicated in the “Dyn” column of Table 2.

3.2. GPP TDG Transformations

Here we describe GPP transformations, which construct a particular GPP’s TDG by transforming another. We first describe how we elide (rip-up) particular edges, then how we insert particular GPP’s TDG by transforming another. We first describe common edges across GPP processors, then edges which are specific to OOO and inorder GPPs.

Eliding Edges The first step of GPP transformation is to elide or rip-up edges which need to be modified. These include edges representing architectural features which may be changed, or they could be dynamic edges which do not have defined start and end nodes. There are certainly edges that are not elided, including data and memory dependences, and edges enforcing pipeline serialization.

Common GPP Transforms The GPP representation is quite similar to [11], so we focus on newly proposed components, as indicated in the last column of Tables 1 & 2. Broadly speaking, their model contains edges for representing the issue-width, pipeline dependences, and execution and memory latency. Because we add support for dynamic resources, we can additionally represent functional unit and memory bandwidth resources using dynamic edges. The L1 bandwidth is represented by treating load/store ports as resources, and L2 bandwidth is approximated by considering a finite number of coalescing MSHRs as resources. Note that the memory inter-

<table>
<thead>
<tr>
<th>Table 2: Inter-Inst. Edges (Fx:Fixed, Cm:Computed, Rc:Recorded)</th>
<th>Edge Constraint</th>
<th>Lat Dyn New</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{i-1} \rightarrow P_i$ Complete In-order</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>$P_{i-1} \rightarrow E_i$ Long latency Inst$_{i-1}$ stalls Inst, by pipe latency</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>$G \rightarrow E_i$ Group for Accel Op, begins before execution</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>$P_{i-1} \rightarrow G_i$ Complete for Accel Op, begins next Group</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>$P_i \rightarrow F_i$ Accel XFER: Complete GPP Inst, before Group</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>$P_i \rightarrow F_i$ GPP XFER: Accel Op, before GPP Fetch of Inst, before Group</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>$E_i \rightarrow F_i$ Enforce issue late for Insts on same FU</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>$P_{i-1} \rightarrow P_i$ Instructions on same FU Complete in-order</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>$E_i \rightarrow P_i$ Data Dep.: Fixed latency models network delay</td>
<td>0</td>
<td>✓</td>
</tr>
</tbody>
</table>
The goal is to maximize the total dynamic instructions captured for some limit of static instructions.

The selection algorithm first builds a set of trees representing the hierarchy of inlineable loops and function calls. The region selection heuristic performs a bottom-up tree traversal, selecting code which has the highest dynamic to static instruction ratio.

**TDG Transform (TDG\textsubscript{GPP,orig} to TDG\textsubscript{GPP,C-Cores})** Inside a C-Cores region of the μDG, according to the analysis plan, this transform elides all fetch, dispatch and commit nodes and edges. Then, since C-Cores only handle one control condition and memory request at a time, a Group (G) node is created for each “basic block,” where they are split to ensure each only contains one memory operation. Edges are added from the Group node to/from the C-Cores instructions, which serialize basic blocks’ execution (see Figure 3 and coproc rows in Table 2).

When the entering/exitng a C-Cores region, edges are inserted to model transfer time. This latency is the amount of live data across regions (computed by analyzing the DFG), divided by the bandwidth between core and accelerator.

### 3.4. BERET TDG

This coprocessor accelerates hot traces of inner loops. It targets energy efficiency by using compound functional units, called Serialized Execution Blocks (SEBs). Diverging from the hot loop trace entails re-execution on the GPP.

**TDG Analysis** The analysis “plan” is a set of eligible and profitable inner loops, and SEB instruction schedules for those loops. Eligible loops with hot traces are found using path profiling techniques [3]. Loops are selected if their loop back probability is higher than 80%, and their configuration size fits in the hardware limit. To eliminate over-specialization to the original target benchmarks, we consider fixed size SEBs, as opposed to specific compound functional units. For scheduling into SEB groups, we use an optimal integer linear program which minimizes the number of register file accesses.

**TDG Transform (TDG\textsubscript{GPP,orig} to TDG\textsubscript{GPP,BERET})** This transform resembles that of C-Cores, where instead of basic blocks, instructions are serialized to execute one at a time.

The GPP interaction is also similar to C-Cores, and uses the same edge insertion algorithm. In addition, if BERET mispredicts the trace path, instructions from that loop iteration are “replayed” on the host processor by reverting to the TDG\textsubscript{GPP,orig} transform (see Figure 3).

### 3.5. 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** The analysis “plan” is a set of legal and profitable loops for vectorization. First, a pass optimistically analyzes the TDG’s memory and data dependences. Memory-dependences 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. 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 (TDGGPP,· to TDGGPP,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 re-mapped 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.

### 3.6. DySER TDG

**DySER** is a reconfigurable circuit switched mesh of FUs, meant for exploiting instruction and data parallelism. It is tightly integrated to the GPP, using custom instructions for communication and a flexible vector interface.

**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 analysis is borrowed from SIMD. Since it only executes computation subgraphs, we use a known slicing algorithm [13] on the loop’s PDG to partition the instructions between the GPP and accelerator. Control instructions which do not have forward memory dependences can be offloaded to DySER.

Similar to SIMD, a “DySER-ized” version of inner loops is considered, where 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 DySER resources, or until the maximum vector length is reached, enabling more parallelism. The analysis algorithm disregards loops with more communication instructions than offloaded computation.

<table>
<thead>
<tr>
<th>Suite</th>
<th>Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPT</td>
<td>conv, merge, nbody, radar, treesearch, vr</td>
</tr>
<tr>
<td>Parboil</td>
<td>cutcp, fft, kmeans, bbm, mm, needle, m, spmv, stencil, tpcdf</td>
</tr>
<tr>
<td>Mediabench</td>
<td>cjpeg, dpjpeg, gsmdecode, gsmencode cjpeg2, dpjpeg2, h264dec, jpeg2000dec, jpeg2000enc, mpeg2dec, mpeg2enc</td>
</tr>
</tbody>
</table>

**Table 3: Benchmarks**

**TDG Transform (TDGGPP,· to TDGGPP,DySER)** DySER keeps a small configuration cache, so if a configuration is not found when entering a DySERized 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 (TDGGPP,· to TDGGPP,SIMD).

Then, DySER-ized instructions are first processed by removing their fetch, dispatch and commit nodes. Then two additional edges to enforce accelerator pipelining: one for the issue width between computation instances (E→E), and one for in-order completion (P→P). We model the scheduling and routing latency by adding a fixed cycle delay on the data dependence edges.

### 4. Infrastructure

We briefly describe some details of our infrastructure in terms of our framework’s implementation and other evaluation choices common to our quantitative results that follow.

**Prism Framework** Creating an accelerator’s TDG in our framework only requires implementing the TDG transform and accelerator-specific analysis algorithms. We emphasize that these algorithms are not application-specific and can be run on arbitrary workloads.

Our framework’s implementation, *Prism*, generates the original TDG using gem5 [4]. We implement custom code for TDG generation, analysis and transformation. Since transforming multi-million instruction traces can be inefficient, Prism uses a windowed approach. Windows are large enough to capture specialization granularity, at most ~10000 instructions for vectorizing loops.

For modeling energy, Prism accumulates energy event counts for both the GPP and accelerator from the TDG. For power/energy, the GPP core activity counts are fed to McPAT [20], a state-of-the-art GPP power model. For fair comparisons, we use McPAT for common components like FUs and reg-files, which it is designed for. For accelerator-specific hardware, we use per-access energy estimates based on existing publications.

**Benchmarks Selection** Benchmarks were chosen from a wide range of suites to match the validation targets, as outlined in Table 3. These include highly regular codes from Intel TPT [13], scientific workloads from PARBOIL [1], image/video applications from Mediabench [19] and irregular workloads from SPECint. The diversity here highlights the
TDG’s capabilities to model specialization in quite different workload environments.

**GPP Configurations** The common GPP characteristics are a 2-way 32KiB IS and 64KiB L1DS, both with 4 cycle latencies, and a 8-way 2MB L2S 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 according to issue width. We use 22nm technology.

**Trace Generation** Dependence traces are generated by fast-forwarding simulation (up to 2 billion instructions), then recording for 200 million instructions.

### 5. Feasibility and Accuracy

Our first quantitative result is that the TDG representation of GPP transformations can accurately model a wide diversity of accelerators. To quantify this, we compare our results from TDG-based modeling of accelerators to previous published results of complete simulator+compiler accelerator models. These results used the original publication’s benchmarks [28, 15, 14].

As an extreme stress test of specialization alone (no program transformation) we also accurately transform a 1-wide OOO processor TDG to represent an 8-wide OOO processor (TDG_{OOO1,8} to TDG_{OOO8,8}). This is a stress test because there are many intricate policies to model, and a simulator that models these in detail typically takes tens of thousands of lines of code. In comparison our GPP TDG transformation is implemented in ~2000 lines of code. For this study, the benchmarks used are an extension of those used to validate the Alpha 21264 [2].

**Accelerator modeling accuracy** Table 4 presents a summary of our validation benchmarks and results.

## Table 4: Validation Results (P: Perf, E: Energy)

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

**OOO modeling accuracy** Figure 4 shows the validation graphs for the OOO core stress test described above (transforming TDG_{OOO1,8} to TDG_{OOO8,8}). The high accuracy here (< 4% avg. error) demonstrates the flexibility of the representation for more or less aggressive hardware.

**Validation Summary** Overall, we achieve an average error of less than 15%, both in terms of performance and energy reduction, compared to simulator or published data.

**Simplicity** Table 5 shows the lines of C++ code required to implement the analysis and the graph transformation for common components and each accelerator. Lines of code is roughly a measure of the “complexity” of a technique. But more importantly, the graph transformations are explicit about how the compiler and accelerator are specializing the processor, and hence are more insightful and easy for designers to use.

### 6. Conclusion

Our work presents a novel program representation of transparent specialization called the TDG, which consists of a closely coupled μDG and Program IR for analysis. This representation allows for the study of the combined effects of compiler and hardware microarchitecture as graph transformations. We showed the TDG approach is practical, feasible and allows deep and insightful analysis.

**References**


