Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor

Ashish Venkat Dean M. Tullsen
University of California, San Diego
{asvenkat,tullsen}@cs.ucsd.edu

Abstract

Heterogeneous multicore architectures have the potential for high performance and energy efficiency. These architectures may be composed of small power-efficient cores, large high-performance cores, and/or specialized cores that accelerate the performance of a particular class of computation. Architects have explored multiple dimensions of heterogeneity, both in terms of micro-architecture and specialization. While early work constrained the cores to share a single ISA, this work shows that allowing heterogeneous ISAs further extends the effectiveness of such architectures.

This work exploits the diversity offered by three modern ISAs: Thumb, x86-64, and Alpha. This architecture has the potential to outperform the best single-ISA heterogeneous architecture by as much as 21%, with 23% energy savings and a reduction of 32% in Energy Delay Product.

1. Introduction

Architects have proposed heterogeneous chip multiprocessors for both general-purpose computing and embedded applications. These architectures exploit heterogeneity in two fundamental dimensions. While some architectures make use of specialized hardware to accelerate the performance of certain workloads [1, 2, 3, 19], others employ a different set of microarchitectural parameters [4, 15, 16, 22, 23, 24] in order to create energy-efficient processors for mixed workloads. The latter constrain the cores to execute a single instruction set architecture (ISA), maximizing efficiency by allowing a thread to dynamically identify, and migrate to, the core to which it is most suited during a particular phase and under the current environmental constraints. This paper demonstrates that not only is that constraint unnecessary, but limiting an architecture to a single ISA restricts the potential heterogeneity, sacrificing performance and efficiency gains.

A critical step in the design of a heterogeneous-ISA architecture is choosing a diverse set of ISAs. While ISAs seem to converge over time (RISC ISAs adding complex operations, CISC ISAs translated to RISC μops internally), there remains sufficient diversity in existing modern ISAs to provide useful heterogeneity. We examine some key aspects that characterize ISA diversity. These include code density, decode and instruction complexity, register pressure, native floating-point arithmetic vs emulation, and SIMD processing.

In this paper, we harness the diversity offered by three ISAs: ARM's Thumb [5], x86-64 [17], and Alpha [12]. By co-designing the hardware architectures and the ISAs to provide the best aggregate architecture, we arrive at a more effective and efficient design than one composed of homogeneous cores, or even heterogeneous cores that share a single ISA.

978-1-4799-4394-4/14/$31.00

The design of a heterogeneous-ISA chip multiprocessor involves navigating a complex search space, made larger by the additional dimension of freedom. A major contribution of this work is such a design space exploration geared at finding an optimal heterogeneous-ISA CMP for general-purpose mixed workloads. Observing the results of the design space exploration, we provide architects with a set of tools to enable ISA-microarchitecture co-design and thereby better streamline their search processes.

To reap the full benefits of the heterogeneity, especially the heterogeneity available in the form of ISA diversity, it is important that an application is able to migrate freely between the cores. However, migration in a heterogeneous-ISA environment is a well known difficult problem [14, 31, 37]. This is because the runtime state of a program is kept in ISA-specific form, and migration to a different ISA involves expensive program state transformation. DeVuyst, et al. [11] demonstrate that migration between ISAs can be achieved at acceptable cost on a CMP; however, that work does not explore the architectural advantages to multiple ISAs on a single CMP. This research employs several ideas from that work, but also several new optimizations to reduce the overhead of migration. In this paper, we present a detailed compilation methodology and an effective runtime strategy that works for a diverse set of ISAs. We observe that even a single application can gain up to 11.2% performance benefit by migrating between heterogeneous-ISA cores during different phases of its execution.

Finally, we evaluate the proposed heterogeneous-ISA CMP against both homogeneous and single-ISA heterogeneous CMPs, under varying power and area budgets. Consequently, we make the following major observations:

- Co-design of ISA and microarchitectural parameters is critical. In the optimal designs, cores employing different ISAs tend to naturally diverge, and to diverge in consistent directions.
- ISA heterogeneity is not only beneficial across applications, but also within individual applications across phases.

We find that heterogeneous-ISA CMPs can improve single-thread performance by an average of 20.8% and provide 15.8% more throughput on multi-programmed mixed workloads, as compared to a single-ISA heterogeneous CMP. Additionally, heterogeneous-ISA CMPs can help achieve an average reduction of 29.8% in Energy Delay Product.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 evaluates the diversity offered by the ISAs chosen for this work. Section 4 lays out our design methodology. We present our compilation and runtime methodologies in Section 5. Section 6 describes our experimental methodology. Section 7 evaluates the proposed

architecture, and draws lessons and guidelines from the design space exploration. Section 8 concludes.

2. Related Work

Prior research has shown that heterogeneous CMPs are capable of higher performance and energy efficiency than homogeneous processors. Single-ISA heterogeneous CMPs were introduced by Kumar, et al. [22, 24]. The motivation behind the single-ISA constraint is that it allows threads to migrate freely between cores dynamically as their behavior or operating conditions change. Recent commercial offerings that resemble this architecture include ARM's big.LITTLE processor [15] and NVidia's Kal-El processor [4].

Yet another class of heterogeneous CMPs make use of specialized hardware to accelerate the performance of certain types of workloads. These include the integrated CPU-GPU architectures such as Intel's Sandy Bridge [1] and AMD's Fusion [2]. However, these architectures do not allow migration between core types at arbitrary places in the code. Current industry offerings of heterogeneous-ISA CMPs include MPSoCs in the embedded market [34], GPUs, and accelerators in the HPC market [3]. IBM's Cell microprocessor [19] is a heterogeneous-ISA CMP geared towards general-purpose computing, but similarly, a too-specialized SPE ISA and lack of a common address space make dynamic task migration infeasible.

Several studies [9, 18, 33] have evaluated the role of ISA in RISC and CISC processors. They show that these ISAs are rather similar in terms of their impact on performance and energy efficiency. However, these studies focus on less diverse ISAs (e.g., PowerPC, ARM, and x86) and homogeneous hardware assumptions. This work differs in two critical ways. First, it examines ISAs with true diversity, and couples heterogeneous ISAs with heterogeneous hardware. We show that there is significant synergy in combining the two, which enables the overall architecture to fully exploit the differences between the ISAs.

DeVuyst, et al. [11] first established the viability of heterogeneous-ISA CMPs by showing that migration cost could be reduced by orders of magnitude over prior approaches, exploiting the shared memory space of CMPs and eliminating the need for memory transfer. That work focuses on the migration mechanism, and does not address any of the architectural implications covered in this work. They examine migration between ARM and MIPS cores, and present compiler techniques to minimize the amount of program state kept in ISA-specific form to enable fast migration. They describe a runtime mechanism that performs binary translation until an equivalence point is reached, after which the program state can be transformed to the required ISA. The Tui system [31] describe a process migration strategy for heterogeneous-ISA machines in the context of wide area computing. The main idea of that work is to transform the runtime program state to an intermediate form and then re-compile it to the required ISA, at the time of migration. We borrow some techniques from both these works; however, our compiler methodology and runtime strategy is geared towards a more diverse set of ISAs, which makes the problem significantly harder and requires additional techniques and optimizations presented in this paper.

Several researchers have proposed design-space exploration methodologies for heterogeneous CMPs. Strozek, et al. [32] describe a process flow for automatic synthesis and evaluation of heterogeneous CMPs based on runtime profiles of certain embedded applications, given different area and power budgets. Intel's QuickIA [10] research prototype allows researchers to explore heterogeneous architectures consisting of multiple generations of Intel processors and FPGAs. The search methodology we employ in this paper is similar to the one described by Kumar, et al. [23]. However, our goal is to not only identify the optimal heterogeneous-ISA multicore designs, but also to lay out the first principles for ISA-microarchitecture co-design in such an architecture.

There has been some early work on operating systems and runtime support for heterogeneous-ISA CMPs. Li, et al. [28] suggest multiple policies and mechanisms, at the operating system level, for symmetric multiprocessing in an overlapping-ISA CMP. For the scope of this work, we assume similar operating system support.

3. ISA Diversity

To keep both the design-space exploration and the compiler development tractable, we select our target ISAs a priori – considering more ISAs and even considering the possibility of custom ISAs would only increase the potential gains. This section describes our three target ISAs – Thumb, Alpha, and x86-64 – with respect to several axes of diversity. These include code density, dynamic instruction count, register pressure, and support for specialized operations.

Code Density. High code density reduces the number of instruction cache misses, uses less energy and memory bandwidth for instruction fetch, and conserves power by enabling the use of smaller microarchitectural structures. Weaver, et al. [38] evaluate a wide range of ISAs for code density. They find that RISC ISAs with fixed-length instructions such as Alpha and SPARC show the lowest code density, while embedded ISAs like Thumb and AVR32 exhibit the highest density owing to a technique called code compression. This technique packs two 16-bit instructions into one 32-bit instruction, which is then unpacked at the decode stage and executed as two instructions. CISC ISAs such as x86-64 and VAX are placed in the middle of the code density spectrum by virtue of variable-length instruction encoding.

Dynamic Instruction Count. While code compression achieves about 32.5% memory savings in Thumb, it increases the dynamic instruction count by 30% [21]. This is a direct consequence of using simpler 2-operand instructions to fit Thumb's 16-bit instruction. Thumb instructions also lack the shift-modifier and predication support that ARM instructions enjoy. Alpha employs 3-operand instructions, but is a load-store architecture, meaning that no arithmetic instruction can directly operate on memory. While x86-64 also restricts in-

structions to the 2-operand format, it implements a number of complex addressing modes that allow instructions to directly operate on memory. x86-64 instructions are decoded into one or more simpler RISC-like μops, thereby increasing the number of dynamic instructions (μops) by about a factor of 1.3 [8] (as compared to the native x86-64 instruction count).

Register Pressure. Thumb uses a reduced register set, allowing only eight 32-bit programmable registers for integer operations. Thus, all 64-bit integer computation is performed using software emulation. Software emulation is discussed in greater detail in Section 5. Alpha, on the other hand, has two banks of thirty-two 64-bit programmable registers, for integer and floating-point computation. x86-64 offers sixteen 64-bit registers for integer operations and sixteen 128-bit registers for floating-point and SIMD operations.

The number of programmable registers is inversely proportional to the amount of register pressure, and thus the number of register spills, for any ISA. Therefore, Thumb suffers from extremely high register pressure, while Alpha enjoys low register pressure. Interestingly, x86-64 enjoys the lowest register pressure among the three ISAs, despite the fact that it has a smaller architectural register file than Alpha. This is a manifestation of the following addressing modes and optimizations: *Absolute memory addressing* allows instructions to directly access memory operands, eliminating the need to allocate registers for temporary storage of loaded values. *Sub-register addressing* allows programmers to address 48 sub-registers to store/operate on smaller data types, which can be further exploited by aggressive sub-register coalescing strategies to reduce the number of register spills. *Program counter relative addressing* enables position-independent code without the overhead (both in performance and allocated registers) of a Global Offset Table. Lastly, *register-to-register spills* allow programmers (compilers) to spill general-purpose registers to XMM registers, thereby minimizing the number of register spills into memory.

Figure 1 shows Thumb instructions and x86-64 μops normalized to Alpha instructions, for the SPEC2006 integer benchmarks, compiled using the LLVM/Clang framework [26, 25]. The average number of dynamic instructions on Thumb is 43.4% more than that of Alpha. This is due, in large part, to the high register pressure in Thumb. In fact,

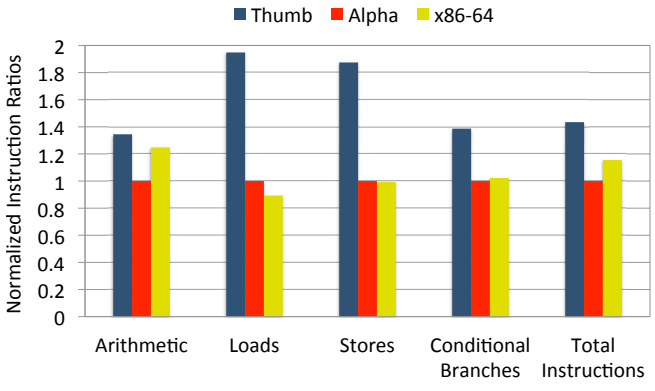

Figure 1: Instruction mix (normalized to Alpha) for SPEC2006

Thumb makes 91.1% more memory references than Alpha. Other factors include 64-bit emulation and the use of simpler 2-operand instructions.

x86-64 makes 8.3% fewer memory references than Alpha due to lower register pressure. However, we observe that there is a very small (0.8%) reduction in the number of stores, while the number of loads drops by 10.8%. The compiler generally seeks to spill variables that won't be updated for a long period of time. Therefore, the number of reads from the spill area is much higher than the number of writes.

Interestingly, Alpha makes 10.9% fewer memory references on the high ILP benchmarks *bzip2* and *hmmer*, in which case the compiler does not find enough opportunity to utilize the complex addressing modes and optimizations offered by x86-64. The 2-operand restriction contributes to the 24.7% more arithmetic instructions on x86-64, resulting in an overall increase in the number of dynamic instructions of 15.4%.

Floating-point and SIMD Support. One consequence of code compression is that floating-point instructions are not supported in Thumb. Floating-point operations are emulated in software. While emulation results in slower execution, Thumb cores don't need to include floating-point instruction windows, register files, and functional units, resulting in up to 19.5% reduction in peak power and 30% savings in area. In a heterogeneous-ISA architecture, any program or phase with significant floating-point activity will likely quickly switch to an ISA that executes natively.

x86-64 also provides SIMD support through its SSE/AVX extensions, making vectorization of loops and basic blocks possible. Alpha's MVI extension allows for only pack, unpack, max, and min operations. Due to the very primitive nature of the MVI extension, we forgo SIMD units in Alpha cores.

To illustrate the benefits of heterogeneity, even on a single application, we examine the performance of *bzip2* during two different phases of its execution (in Figure 2), under varying power constraints. We identify program phases using SimPoint [29]. The detailed methodology is described in Section 6. We see two key results in this graph. First, we see that the most effective ISAs differ between phases of the same application; e.g., at 15 W, Phase 1 prefers x86 and Phase 2 prefers Alpha. Second, we see that even in a single phase, the best ISA varies depending on the design constraints or the operating condition of the processor. For example, in Phase 2, we might prefer Alpha unless we are operating unplugged or perhaps in a low battery state, in which case we'd prefer Thumb because at low power budgets, Thumb provides the highest performance.

4. Design Space Exploration

The possible design space of a heterogeneous-ISA CMP is characterized by a diverse set of ISAs and a multitude of microarchitectural parameters. Navigating such a design space is a difficult problem. That difficulty can be reduced and pruned if we understand some of the principles that govern the effective co-design of heterogeneous-ISA, heterogeneous hardware architecture processors. To do this, we execute an exhaustive design space exploration of an architecture with

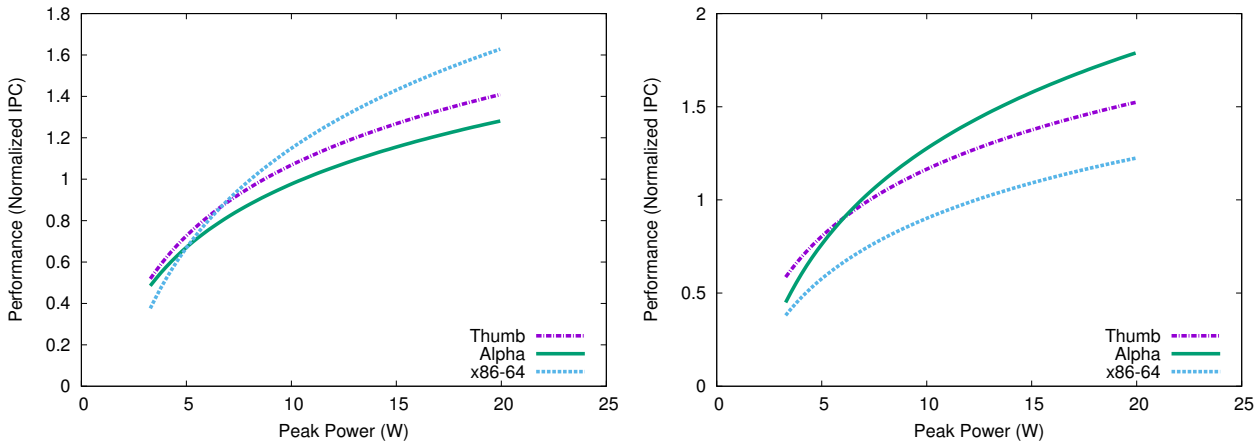

Figure 2: Performance comparison under different peak power budgets for two different execution phases of bzip2

Design Parameter	Design Choices
ISA	Thumb, Alpha, x86-64
Execution Semantics	In-order, Out-of-order
Issue width	1, 2, 4
Branch Predictor	local, tournament
Reorder Buffer Size	64, 128 entries
Architectural Register File	ISA-specific
Physical Register File (Integer)	96, 160
Physical Register File (FP/SIMD)	64, 96
Integer ALUs	1, 3, 6
Integer Multiply/Divide Units	1, 2
Floating-point ALUs	1, 2, 4
FP Multiply/Divide Units	1, 2
SIMD Units	1, 2, 4
Load/Store Queue Sizes	16, 32 entries
Instruction Cache	32KB 4-way, 64KB 4-way
Private Data Cache	32KB 4-way, 64KB 4-way
Shared Last Level (L2) Cache	4-banked 4MB 4-way, 4-banked 8MB 8-way

Table 1: Design space of a Heterogeneous-ISA architecture

fairly limited, tractable options, and observe the characteristics of the best designs.

The design space we explore in this work includes the three ISAs - Thumb, Alpha, and x86-64, along with a set of micro-architectural parameters that represent a wide range of performance and power control points. The goal of the design-space exploration is to find the optimal 4-core heterogeneous-ISA CMP for varying power and area budgets, and considering all permutations of applications in the workload sharing the cores.

Table 1 enumerates the variables in our design space. The Cartesian product of this design space consists of 750 thousand single core combinations, making it not practically feasible to perform an exhaustive search. To reduce the size of the Cartesian product, we prune the design space by establishing correlations between different variables. While some correlations can be inferred by intuition, others are dictated by specific ISA characteristics: (1) Size of the reorder buffer can be correlated to the physical register file size, as they together establish the window size. (2) The number of functional units varies with the issue width. (3) Thumb cores need not include a floating-point instruction window, retirement units, register files or functional units. (4) Neither alpha nor thumb need include SIMD functional units.

Design Parameter	Design Choices
ISA	Thumb, Alpha, x86-64
Execution Semantics	In-order, Out-of-order
Branch Predictor	local, tournament
Reorder Buffer-Register File	64-96-64, 128-160-96 entries
Issue Width-Functional Units	1-1-1-1-1-1, 1-3-2-2-2-2, 2-3-2-2-2-2, 4-3-2-2-2-2, 4-6-2-4-2-4
Load/Store Queue Sizes	16, 32 entries
Cache Hierarchy	32K/4-32K/4-4M/4, 32K/4-32K/4-8M/8, 64K/4-64K/4-4M/4, 64K/4-64K/4-8M/8

Table 2: Pruned design space for faster navigation

The resulting pruned design space, as shown in Table 2, contains 120 in-order cores and 480 out-of-order cores. However, the number of possible 4-core configurations in the pruned design space is still very high (129.6 billion configurations). To further make this problem tractable, we use the following results from prior research on single-ISA heterogeneous architectures – modeling using private LLCs versus a shared n-banked LLC in an n-core configuration, results in the same performance ordering with respect to all n-core configurations [23], as well as different scheduling/migration strategies [35]

Therefore, we model cores using 1MB 4-way or 2MB 8-way private last-level caches, instead of a single shared 4MB or 8MB cache, respectively. Thus, the combined performance of a 4-core configuration with private LLCs can be computed using the sum of the performances of the individual cores. While the design space exploration still involves finding the optimal 4-core configuration out of 129.6 billion different configurations, we can now find the best design with 600 simulations of the single-core permutations, and a software search of the 130 billion sums.

5. Compiler and Runtime Environment

The programming environment for a heterogeneous CMP is dictated by one of the first design choices: separate address space [3, 4] vs unified address space [15, 22]. We contend that the full benefits of a heterogeneous multicore architecture can be reaped only through dynamic core selection, which requires process migration. Separate address space constraints impose

a significant cost to process migration in terms of program state transfer. Therefore, we choose the unified address space model in our design. However, this presents a unique challenge to compilation and process migration, because the memory layout and runtime state of a program is always architecture-specific.

To attack the process migration problem, we borrow a number of compiler and runtime techniques from prior work by DeVuyst, et al. In particular, our compiler generates a *fat binary* with multiple target-specific code sections and common target-independent data sections. Furthermore, our runtime environment employs dynamic binary translation till we reach an *equivalence point*, at which program state can be successfully transformed. However, the ISAs we have chosen for this work are significantly more diverse and therefore present additional challenges that are not just limited to compiler and runtime support.

First, we deal with both 32-bit and 64-bit ISAs that each expose a different organization of the virtual address space and page table hierarchies. This necessitates the use of a common address translation scheme and long mode emulation on the 32-bit Thumb ISA. Second, the complexity of creating common target-independent data sections increases with both the number and diversity of ISAs. We take a cleaner approach here by using a common intermediate representation and encapsulate the lowest common denominator size and alignment rules in intermediate-level types. Third, to reap full benefits of ISA heterogeneity, it is critical that we don't turn off any target-specific compiler optimization. Therefore, unlike prior work, we can no longer rely on a map of source-level variable names to transform every live CPU register or memory location, at the time of migration. We instead take advantage of a powerful architecture-independent intermediate representation that can act as a bridge between the ISAs, and provide hints for faster and more frequent program state transformation. As a result, in this work we introduce a compilation infrastructure that is significantly more flexible (capable of working with more diverse ISAs) and provides higher performance, both in steady state (native mode execution) and at migration points.

In this section, we first present a memory management strategy to facilitate a unified address space, and then describe a compilation and runtime strategy to address additional challenges presented by heterogeneous-ISA process migration.

5.1. Memory Management

Address Translation. A common address translation mechanism is required to ensure a unified address space in a heterogeneous-ISA environment. From Table 3, Alpha emerges as the lowest common denominator due to its 8KB page size. While it is possible to use software virtualization for 8KB page management on Thumb and x86-64, it necessitates the use of multiple page table structures, one for each ISA. Furthermore, software virtualization cannot enable 64-bit virtual address translation on the 32-bit Thumb architecture. Therefore, we use a common page table structure and an MMU based on the 4-level page table walker of x86-64, for all the three ISAs.

ISA	Thumb	Alpha	x86-64
Page Table Hierarchy	2-level	3-level	4-level
Page Size	4KB, 64KB	8KB	4KB, 2MB, 1GB
Page Table Size	16KB first-level, and 1KB second-level	8KB	4KB
TLB Update Mechanism	Hardware page table walker	Low-level firmware (PALcode)	Hardware page table walker

Table 3: Memory Management on Thumb, Alpha and x86-64

Long mode emulation on Thumb. Long mode (64-bit) computation in Thumb is performed using software emulation. Most compilers already support this to perform arithmetic and memory operations on the "long long" data type. The general procedure is to use multiple 32-bit registers to construct 64-bit values and compute on them.

To support memory operations using 64-bit pointers (virtual addresses), we extend the Thumb ISA to include special instructions: LD64 and ST64. These instruct the MMU to look for the higher-order 32-bits of its 64-bit virtual address input in a special register *R8*. However, the memory footprint of most general-purpose workloads seldom exceeds 4GB. In fact, we observe that no SPEC CPU2006 benchmark acquires more than 4GB of memory. Therefore, wherever possible, we use the regular load/store instructions with 32-bit pointers, which are zero-extended by the MMU during address translation.

5.2. Compilation Strategy

Common Intermediate Representation. In this work, we leverage the LLVM compiler framework [26] and the Clang front-end [25] to generate a common intermediate representation (LLVM bitcode), and perform target-specific backend compilation thereafter. To keep the front-end compilation ISA-agnostic, we make use of the *target-triple* functionality of Clang to specify the data types of a generic target, for all ISAs.

Target-Independent Type Legalization. To minimize the amount of program state to be transformed, we enforce common rules for promotion, truncation, expansion, and type conversion. We allow certain exceptions during type legalization that interfere with ISA diversity – e.g., vector widening/scalarizing on x86-64, and long mode/floating-point emulation on Thumb. For the most part, target-independent type legalization ensures a consistent view of global data and bitcode-level variables across all ISAs, during every stage of compilation. This is critical for generating a single version of target-independent global data sections.

Intermediate Name Propagation. Once the intermediate representation has been generated, we provide each bitcode-level variable with a unique name. During the subsequent code generation and optimization passes, we ensure that each target-level machine operand (both registers and fixed stack slots) is associated with its corresponding intermediate name, if any. This gives us the ability to distinguish between bitcode-level and target-level variables, which plays a key role at the time of transform generation.

Stack Frame Organization. To avoid handling pointer inconsistencies at the time of program state transformation,

we enforce a common stack frame organization (see Figure 3). In doing so, we do not add any additional instructions, since we at most change the relative position of a stack object from the stack/frame pointer. To ensure stack consistency across ISAs, we use similar techniques as described by DeVuyst, et al. [11].

Generation of Runtime Transforms. Towards the end of the multi-ISA compilation, we generate a set of transforms that can be applied by the runtime environment, at the time of migration. Each transform is a routine that reconstructs the target-specific program state of a basic block (both live registers and stack objects), in the required ISA-form. Accordingly, a live register or a stack object can be transformed if one of the following conditions hold:

- Its value is known at compile time - e.g., constant literals.
- Its value can be found at a specific location - e.g, globals, immutable objects (aggregates, alloca variables, and variables whose addresses have been taken).
- Its intermediate name refers to a live register or stack object on the ISA from which we migrated.
- Its value can be computed using the already transformed live registers and stack slots. This involves a reverse traversal of the def-use chain to find a sequence of instructions that can re-compute the required value.

We generate transforms at compile-time rather than runtime because performing reverse data-flow analysis to determine the value of every live register and stack object imposes a significant cost to migration. Prior research [11] suggests that we can achieve fast stack transformation by just matching the live register and stack contents between the ISAs involved, and copy-in their values accordingly. We found that such a strategy results in very few transformable basic blocks, if the ISAs exhibit significant diversity. Specifically, we observed that the transforms derived using the reverse data-flow analysis reduced the run-time distance to the next transformable basic block by an order of magnitude.

Theoretically, it is possible to transform any live register and stack object in the program, using such a reverse data-flow analysis. However, due to the limited amount of information available at compile-time, we cannot handle program constructs such as uncountable loops, indirect branches, and conditional branches. Therefore, we restrict this analysis to control flow graphs that do not contain ϕ-nodes (merge points). Note that ϕ-nodes could still be transformable using methods described above, other than reverse data-flow analysis.

Runtime Environment. Our runtime environment has two major components: a dynamic binary translation (DBT) engine and a program state transformer. The DBT engine itself consists of six individual translators, based on the *tiny code generators* of QEMU [6]. At the time of migration, dynamic binary translation is performed until an *equivalence point* is reached, after which it is safe to use the compiler-generated transforms to convert program state from one ISA to another. Each compiler-generated transform is recursive in nature. After transforming the program state of the current basic block, it walks up the stack using the return address and invokes the caller's transform. On their return path, the transforms fix up the callee-saved register spill area, which ultimately culminates in the construction of the target CPU register state.

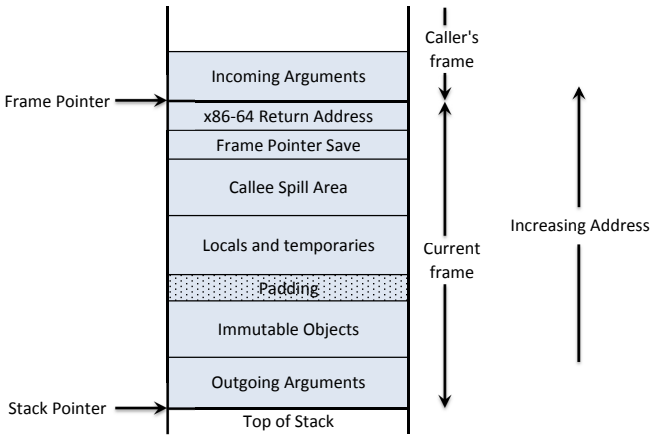

Figure 3: Common Stack Frame Layout across all ISAs

6. Experimental Methodology

In this section, we describe our experimental methodology for the two major contributions of this work: (1) design space exploration for a heterogeneous-ISA CMP, and (2) compiler and runtime strategy for a diverse set of ISAs.

6.1. Design Space Exploration

Our four-core design space consists of 600 homogeneous processors, 1.5 billion single-ISA heterogeneous processors, and 128.3 billion heterogeneous-ISA chip multiprocessors, that can be each designed out of 600 distinct CPU cores.

Although prior work on ISA characterization [9, 18, 33] chooses to model cores based on actual commercial offerings for each ISA, we seek to remove any non-ISA biases and start each design with a clean slate. Thus, we assume the same basic pipeline design (number of stages and latency), based on the Alpha 21264 [20], across all ISAs (with the exception of instruction decode, which will be the primary stage(s) that depend on the ISA). Additionally, we assume a total-store-order (strictest of all) memory consistency model for all ISAs.

All cores are modeled using 32nm technology and the clock rate is fixed at 1.67GHz. Owing to ISA diversity and the multitude of micro-architectural parameters we consider in this work, the heterogeneous-ISA CMPs are distributed over significant peak power (8.32-80.69 W) and area (32.97-129.87 mm^2) ranges. We use the gem5 [7] simulator to model CPU core performance, and McPAT [27] to model power and area.

Our design methodology selects the optimal multicore configuration over the entire set of workloads (all possible permutations), for different peak power and area budgets. The design space explorations are optimized for two types of workloads: (1) multi-programmed mixed workloads, and (2) single-threaded workloads. The former helps us evaluate the throughput of a conventional CMP, the latter gives us insight into a "Dark Silicon" implementation, where it is expected that only one core (out of a heterogeneous cluster) will be powered up at once [13, 22, 36]. In the latter case, a thread will always

be assigned it's best core, but in the former case, it will depend on the threads with which it is co-scheduled. We will also examine both the case where threads are placed based on overall execution characteristics (assuming minimal migration), and the case where threads can migrate to other cores at phase changes. That is, in the first case we find the best assignment of applications to cores, in the second, we find the best assignment of phases to cores.

We use SimPoint [29] to identify program phases. Specifically, we obtain multiple simulation points for a program's execution on Alpha, with an interval size of 100 million dynamic instructions. We modify the *atomic* CPU (instruction emulation mode) of gem5 to emit the start and end basic blocks for each simulation point, and their cumulative frequency, which serve as the start and end markers for the corresponding program phase on the other two ISAs, namely thumb and x86-64. Our workloads include a total of 72 different program phases on the 10 applications we benchmark.

6.2. Compiler Methodology

We use the SPEC CPU2006 integer and floating-point C benchmarks to evaluate the proposed architecture. We exclude *h264ref* and *perlbench* from this set because they use ISA-specific programming constructs (e.g., inline assembly), either directly or through library function calls. All benchmarks are compiled at the -O3 optimization level, using the multi-ISA compilation methodology described in section 5. We perform all experiments that evaluate the performance of our compiler and runtime methodology on cores that are modeled after Cortex A-15 for Thumb, the 21264 processor for Alpha, and Core-i7 for x86-64. We use the following metrics to evaluate our compilation methodology.

Steady State Performance. The primary goal of multi-ISA compilation is to enable seamless execution migration between different ISAs at minimal cost. Through consistent compilation and deterministic transform generation, we manage to be migration-safe in 45% of the basic blocks. However, since we enforce many consistency rules during the multi-ISA compilation, it is important to study its effect on steady-state performance. To evaluate the steady-state performance degradation, we simulate each program phase of a benchmark compiled for both single-ISA execution and multi-ISA execution.

Distance to Next Equivalence Point. This distance represents the number of dynamic instructions to be translated by the DBT engine, before we can transform the program state to enable native execution. We modify gem5's *atomic* CPU to find if a given instruction is an equivalence point, using compiler metadata. We then run each benchmark to completion and record the average distance to the next equivalence point.

Migration Cost. Migration cost consists of two major components: dynamic binary translation and program state transformation. On every ISA, we take 10 samples of the benchmark's dynamic execution state, each at a 100 million instruction interval, after fast-forwarding execution for the first one billion instructions [30]. We then simulate heterogeneous-ISA migration scenarios for each sample, by performing dynamic binary translation until an equivalence point is reached, and program state transformation at that point.

Overall Speedup Due to Migration. To compute the overall speedup due to migration, we employ a phase based scheduling strategy, where migration happens only when phase transitions demand switching to a different core. To identify phase transitions, we rely on SimPoint metadata and profiling information from oracle experiments. This models a system where the compiler is directing migration (or at least migration preferences), or a runtime or hardware system that had been observing execution long enough to accurately detect phase behavior.

7. Results

This work seeks to identify the best heterogeneous designs for a given workload. This not only enables us to quantify the potential gains for ISA heterogeneity, but also identify trends and insights from the actual designs that get tagged as optimal. Because the nature of the design exploration is relatively independent of the cost of migration, only results later in this section account for the specific costs of migration, and the extent to which they mitigate the potential gains.

7.1. Evaluation of the Heterogeneous-ISA Architecture

Processor designs today are as likely to be constrained by power dissipation as they are by area. Thus, in this section, we examine the top-performing designs under both area and power constraints. In addition to finding the best heterogeneous-ISA design, we also find the best homogeneous design (best single configuration for any ISA) and best single-ISA heterogeneous design (best heterogeneous design for which all cores have the same ISA). We consider designs optimized for both multi-programmed workload throughput and single-thread performance.

Multi-programmed workloads. Figure 4 compares three architectures: homogeneous, single-ISA heterogeneous, and heterogeneous-ISA CMPs, all optimized for multi-programmed workload performance under different peak power and area constraints. We make several important observations here. First, there are significant gains available from ISA heterogeneity, matching or exceeding the gains from hardware heterogeneity. Second, hardware heterogeneity alone is less effective under tight constraints (all cores have to be small) or liberal constraints (all cores free to be big), because both endpoints tend toward homogeneous designs. Heterogeneous-ISA designs, in contrast, are still effective in those regions, because we can still gain from ISA heterogeneity even when the hardware is homogeneous.

There are two reasons for this advantage. First, different code regions have a natural affinity for one ISA or another, irrespective of the hardware implementation. Second, the ISA options give the architect more opportunities to create area-effective or power-effective cores.

For instance, at a peak power budget of 20W, the single-ISA heterogeneous CMP manages to employ only 3 out-of-order cores with smaller 32KB L1 caches, while the heterogeneous-ISA CMP sports all out-of-order cores with 64KB L1 caches.

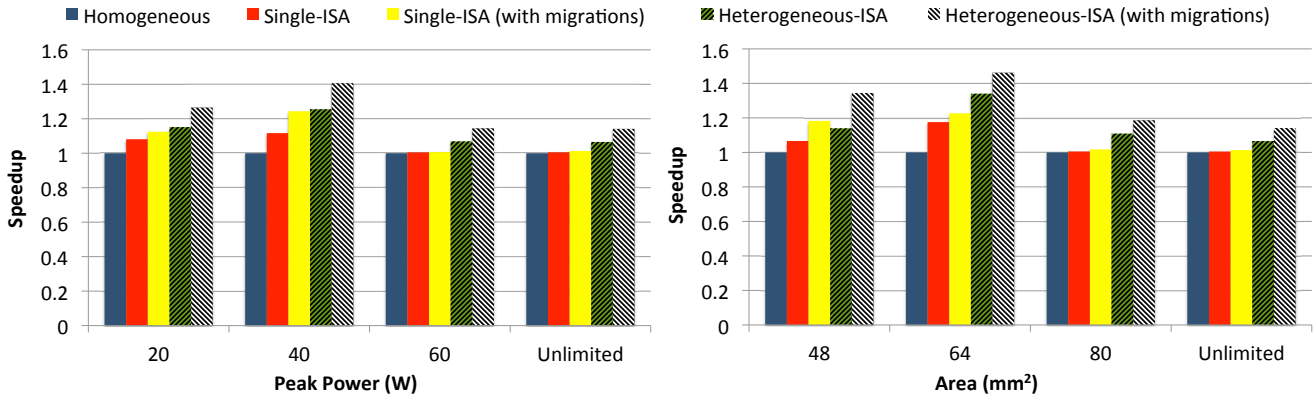

Figure 4: Multi-programmed Workload Performance comparison under different peak power and area budgets

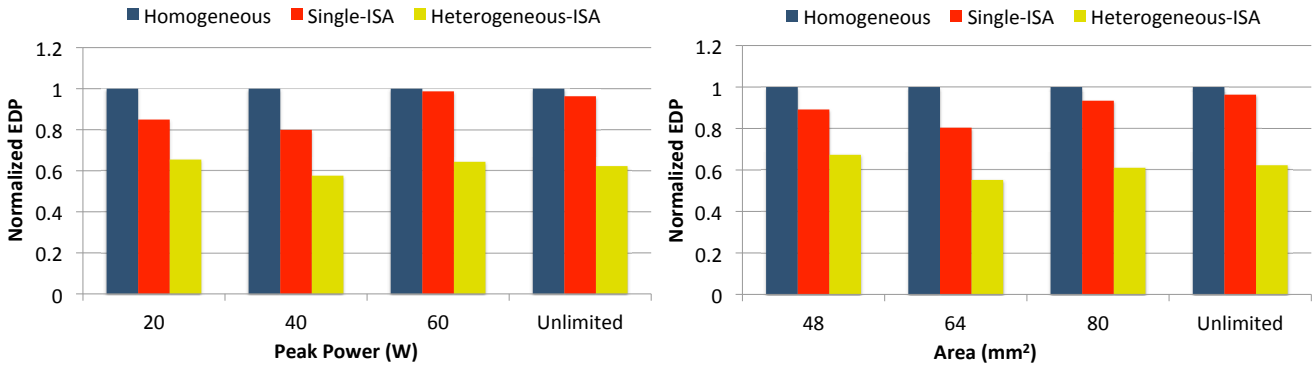

Figure 5: Energy-Delay-Product comparison for multi-programmed workloads under different peak power and area budgets

This is possible because the area-efficient and power-efficient Thumb cores free up space that is put to good use by the other cores. We find that heterogeneous-ISA CMPs can provide 15.8% better throughput on multi-programmed workloads than the best single-ISA heterogeneous CMPs.

Furthermore, we can achieve a greater speedup if applications are allowed to migrate between the cores at phase boundaries. This is well-documented in the case of single-ISA heterogeneous CMPs [22, 24]. On a heterogeneous-ISA CMP, this effect is further enhanced due to ISA affinity. We observe an additional speedup of 11.2% due to migration alone on a heterogeneous-ISA CMP, in contrast to the 4.6% speedup due to migration on a single-ISA heterogeneous CMP. In all subsequent experiments in this paper, migrations are always enabled.

In order to evaluate energy efficiency, we instead optimize the design space exploration to find energy-efficient cores by identifying the processor configurations that minimize energy-delay product (EDP). Figure 5 compares the energy efficiency for the three architectures under different peak power and area budgets. Heterogeneous-ISA CMPs achieve an average energy savings of 21.5% and an average reduction of 27.8% in the EDP over single-ISA heterogeneous CMPs, with absolutely no loss in performance – that is, we gain performance and decrease energy simultaneously when we employ multi-ISA heterogeneity.

Thus we see that the energy efficiency gains of ISA heterogeneity actually exceed the potential performance gains (for the performance-optimized experiments). Maximizing heterogeneity in this way can be particularly effective in a power-constrained environment. In a homogeneous-ISA general-purpose processor, Thumb is not a serious candidate, because it performs so poorly for certain codes; however, as part of a heterogeneous solution, it shines for certain code regions.

Single-threaded workloads. To evaluate designs optimized for single-threaded workloads, under different peak power budgets, we assume the *dynamic multicore* topology described by Esmaeilzadeh, et al. [13], in which idle cores are turned off to reduce power consumption. Figure 6 shows performance and EDP measurements for the three architectures constructed when searching the design space for multicore architectures that provide optimal performance or energy efficiency over our benchmark set. We apply lower peak power constraints in this scenario, since we are optimizing for the single-powered-core execution scenario.

We observe that in a highly peak power constrained environment, the heterogeneous-ISA CMP still manages to achieve a speedup of 16.9% over a single-ISA heterogeneous CMP, and as the peak power budget becomes slightly higher (at 15 W), it provides a consistent speedup of 18.8%. However, the maximum speedup that a single-ISA heterogeneous CMP can provide over a homogeneous CMP in such a topology, is just 1.5%. So again we see that ISA heterogeneity continues to provide gains in regions where hardware heterogeneity is less effective.

Figure 7 shows the performance and EDP evaluation on de-

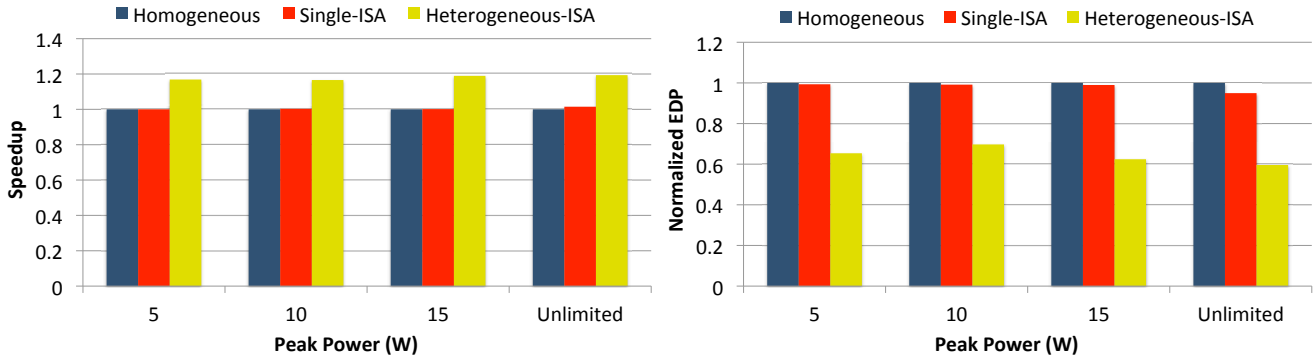

Figure 6: Single Thread Performance and EDP evaluation using the dynamic multicore topology

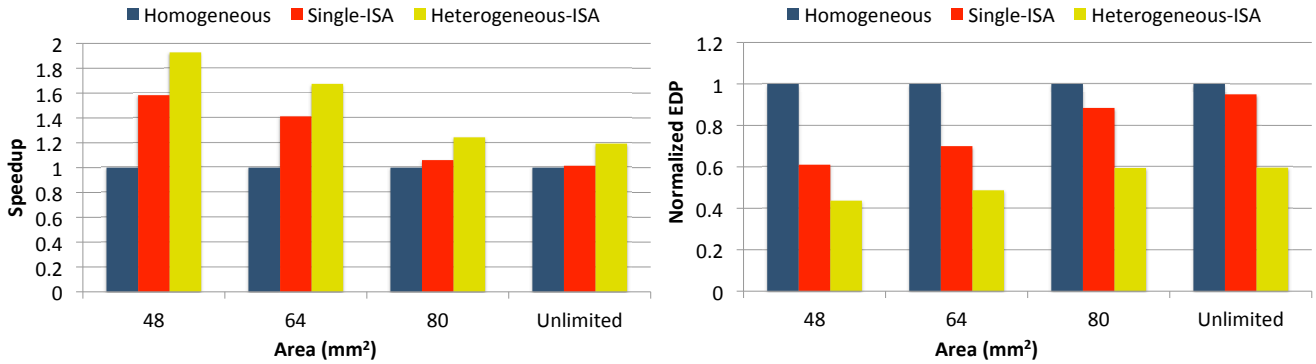

Figure 7: Single Thread Performance and EDP evaluation under different area budgets

signs optimized for single-threaded workloads, under different area budgets. Such designs are typically composed of multiple small cores and one large core optimized to provide high single thread performance. When highly power constrained, single-ISA heterogeneous CMPs use three small Alpha in-order cores and one powerful out-of-order core. Combining again the dual benefits of ISA affinity and the area benefits of Thumb, the best heterogeneous-ISA CMPs provide more balanced cores (to better exploit ISA affinity) yet still enable the same large Alpha core as the single-ISA design. That configuration contains two small Thumb cores, the same out-of-order Alpha core, and a medium-end x86-64 core. We observe that a heterogeneous-ISA CMP can improve single-thread performance by 20.8% over a single-ISA heterogeneous CMP, or achieve 23% more energy savings and 31.8% reduction in EDP, again with no loss in performance.

7.2. Framework for ISA-Microarchitecture co-design

In this section, we present inferences from our design space exploration that can serve as a framework for future ISA-microarchitecture co-design in the context of a heterogeneous-ISA CMP. We consider the best designs from all experiments carried out in the previous section. Figure 8 shows the frequency of occurrence of different micro-architectural parameters in a heterogeneous-ISA design. We analyze the influence of ISA on each of these micro-architectural parameters.

Execution Semantics. We find that out-of-order execution semantics is favorable in general. However, due to conservative peak power budgets in some designs, we select inorder

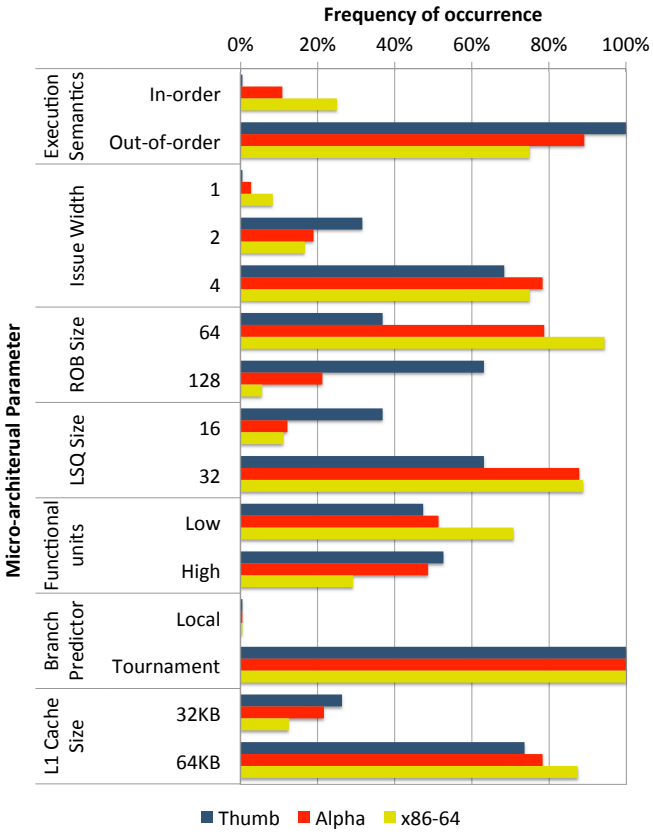

Figure 8: Inferences from the Design Space Exploration

cores 10.8% of the time on Alpha, and 25% of the time on x86-64. Due to the enormous peak power and area benefits of Thumb, we always select out-of-order Thumb cores because they are so cheap. This impacts a number of our results, because it means even our smallest designs always have an out-of-order core available.

Issue Width. In general, higher fetch width enables higher issue width. Since the instruction fetch units of Thumb and Alpha dissipate less power than x86-64, we seldom choose a small issue width for these ISAs. In fact, only 2.7% of our designs choose a single-issue Alpha core and none of our designs use a single-issue core for Thumb. Because code compression ensures our minimum fetch bandwidth is 2 instructions with Thumb, a scalar Thumb processor would always have fetch and issue out of balance.

ROB Size. We find that the number of ROB entries and register file size are highly influenced by the register pressure of an ISA. The low register pressure of x86-64 (see Section 3) results in small ROBs and physical register files being configured. Conversely, the high register pressure of Thumb has the opposite effect, while Alpha finds a middle ground between the two.

Load/Store Queue Size. Although we do not see a direct correlation between load/store queue sizes and ISA traits, ISAs more likely to be configured out-of-order and with wide issue, not surprisingly, also demand large LSQs.

Number of Functional Units. Because the x86-64 cores we model have more basic functional unit types (integer, floating-point, and SIMD), the cost of going from the low to the high configuration is higher, and that step is taken less often.

Branch Predictor. Interestingly, all ISAs almost always choose the tournament branch predictor. This implies that it is always worthwhile to invest transistors on the branch predictor. ISA traits such as predication support have little influence on the selection of branch predictor type.

L1 Cache Size The size of L1 cache largely depends on the working set of applications, rather than a specific trait of an ISA. In general, all ISAs favor higher L1 cache sizes.

7.3. ISA Affinity

To determine the ISA affinity of each application, we simulate both single-threaded and multi-threaded workloads on two types of designs: (1) optimized for performance, and (2) optimized for EDP. In all scenarios, designs are constrained by a peak power budget of 40 W. Each scenario provides some interesting insights. The design optimized for single-threaded performance is the true indicator of ISA affinity, since only one application is in execution at a time, and each application is allowed to freely migrate between the cores. In case of multi-programmed workloads, due to contention between applications, some applications may execute on ISAs of second preference. On designs optimized for energy efficiency, applications may choose to execute on ISAs that provide energy efficiency but don't maximize performance.

Figure 9 shows that each application exhibits a different degree of ISA affinity, and most use all ISAs. In our experiments, benefits arise due to a combination of ISA factors, some synergistic, some interacting negatively – trying to separate those effects is difficult and not always intuitive. However, we are able to make a few high level observations. (a) No floating-point benchmark prefers execution on Thumb due to floating-point emulation. (b) The floating-point benchmark *lbm* prefers execution on Alpha instead of x86-64, because Alpha requires about 34% fewer dynamic floating-point instructions. (c) The high ILP benchmarks *bzip2*, *hmmer*, and *sjeng* prefer execution on Alpha over x86-64, because Alpha offers lower register pressure during phases of high instruction-level parallelism (see Section 3). (d) *bzip2* prefers execution on the 32-bit Thumb ISA during phases that involve 32-bit unsigned integer arithmetic. Alpha incurs 27% more dynamic instructions to emulate 32-bit arithmetic using 64-bit registers. In such phases, x86-64 emerges as the ISA of second preference due to sub-register addressing. (e) The benchmarks *libquantum*, *milc*, and *sphinx3* take advantage of x86-64's SIMD functionality at different execution phases, and revert back to Alpha/Thumb during the scalar phases.

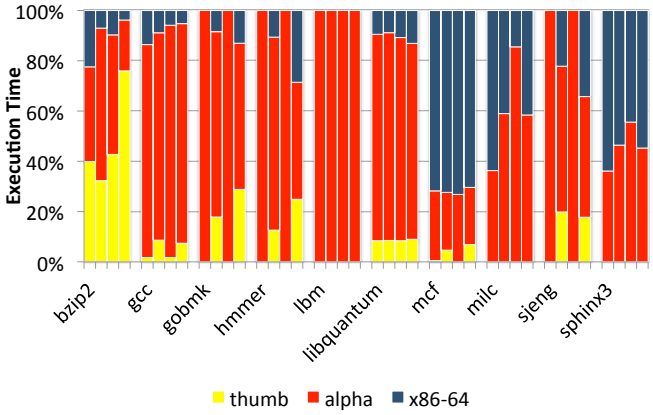

Figure 9: ISA affinity for different applications on designs optimized for (left to right) - (a) Single-thread performance, (b) Multi-programmed workload performance, (c) Single-threaded workload EDP, (d) Multi-programmed workload EDP

Not immediately clear from the results so far is to what extent the gains are a result of broad differences in feature sets (e.g., SIMD vs no SIMD support) as opposed to the more subtle differences. Further experiments show that the former are a surprisingly small component. For example, if we consider x86-64 with vs without SSE, that heterogeneity provides a gain of 1.3% over the best single-ISA configuration – significantly lower than the 15.8% speedup from a fully heterogeneous-ISA design.

Finally, we note that there is little deviation in ISA affinity due to contention amongst multi-programmed workloads, or due to optimization for EDP instead of performance.

7.4. Compiler and Runtime Evaluation

The prior results, primarily concerned with the discovery of the best core configurations, do not account for the cost of reduced compiler effectiveness (which would affect steady state performance) or of migration itself. Each of these would

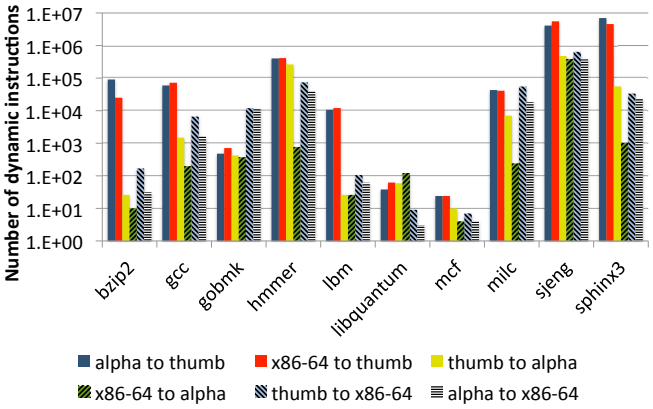

Figure 10: Number of dynamic instructions to be translated before program state can be transformed

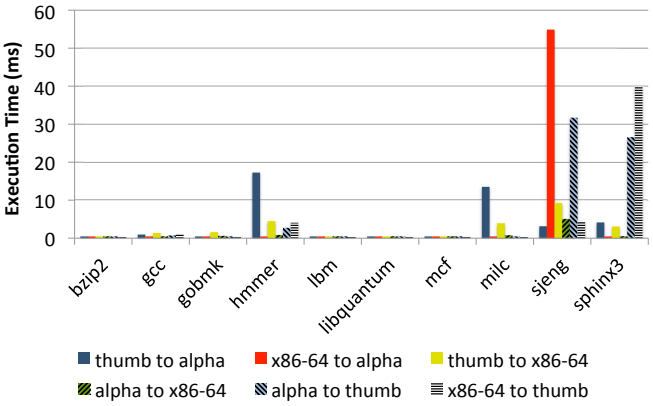

Figure 11: Average Migration Costs (includes both Binary Translation and Program State Transformation)

degrade performance or efficiency gains over a single-ISA design (where migration is much faster). This section evaluates those costs.

Steady State Performance. This examines the cost of multi-ISA compilation on regular execution, compared to single-ISA compilation. Due to the long-mode emulation on Thumb, we observe a 4.6% average loss in performance; however, that does not actually impact our results – for code that incurs that emulation, we typically do not select the Thumb core for execution. x86-64 and Alpha show no performance degradation at all, up to four decimal places of the IPC. Such small degradation of performance comes from the fact that we do not disable any optimization or ISA-specific behavior to enable multi-ISA compilation. This is in contrast to the prior work [11] which sacrifices 2-3% for multi-ISA compilation.

Fat Binary Overhead. The fat binary created by our compiler contains multiple code sections, but a core only loads its own code to its private Icache. In fact, this can only impact a shared cache. If we change our design to have a shared 16MB LLC, and incorporate the increased working set size, we measure zero performance loss. At a 40 W peak power budget, the best heterogeneous-ISA design achieves 47.5% savings in instruction fetch energy over the best single-ISA heterogeneous design, due to code density advantages.

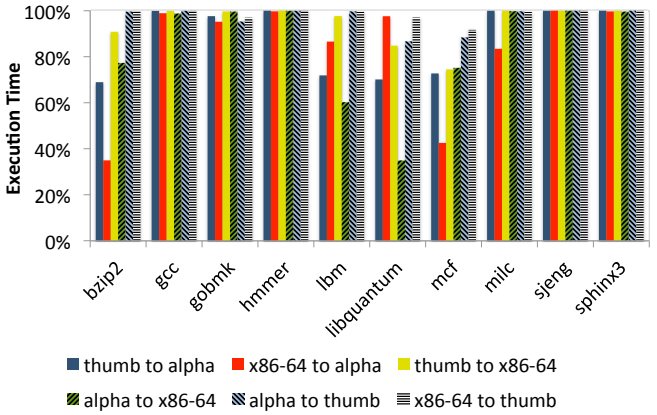

Figure 12: Percentage make up of Binary Translation Cost in the Overall Migration Overhead

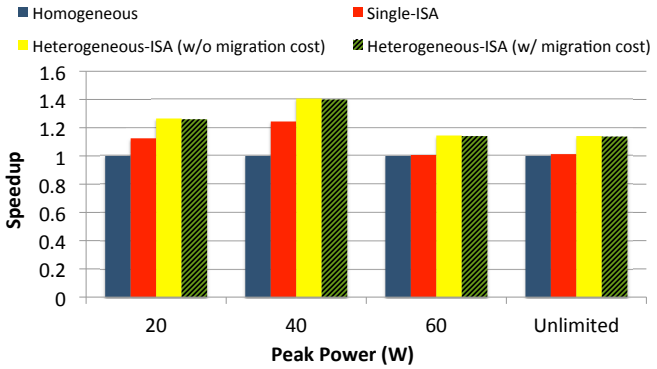

Figure 13: Overall Speedup due to migration.

Distance to Next Equivalence Point. Figure 10 shows the average distance to the next equivalence point for all possible migration scenarios. Due to the high frequency of equivalence points for both Alpha and x86-64, fewer number of instructions (in the order of tens of thousands) are needed to reach an equivalence point. On the other hand, migrating to Thumb can require binary translation of millions of instructions before an equivalence point is reached.

Migration Cost. Figure 11 shows the total overhead per migration, assuming migration at random intervals in execution. We measure an average overhead of 4 milliseconds to switch between ISAs, but that is heavily influenced by a few outliers. In most cases, average migration cost is insignificant if we assume we are not migrating more often than every couple hundred milliseconds. The migration overhead shows high correlation to the distance to reach an equivalence point. This implies that the migration overhead is typically dominated by binary translation. Figure 12 shows the percentage of time spent in binary translation during migration. For six out of ten benchmarks, this number is more than 90%. The rest of them have a very small migration cost (less than 100 microseconds), and therefore binary translation is not a significant contributor. We also note that our binary translator is not yet heavily optimized for performance.

Overall Speedup due to Migration. In this experiment, we account for the cost of the actual migrations encountered in an earlier experiment. That is, we incur the cost of migration

between two ISAs when a phase change causes a new core/ISA combination to be preferred. Figure 13 shows the result of this experiment. Here we see that we sacrifice negligible performance (about 0.4-0.7%) for migration, meaning that virtually all of the performance gain from heterogeneous ISAs is retained. This comes from two factors. First, in most cases, migration overhead is very low. Second, phase changes are relatively infrequent, infrequent enough that even our few cases of high migration overhead are not significant.

8. Conclusion

This research explores the design space of heterogeneous-ISA chip multiprocessors. It shows that adding an extra axis of heterogeneity by considering multiple ISAs significantly increases the performance and energy efficiency of a heterogeneous processor. Specifically, a heterogeneous design that allows cores with distinct ISAs outperforms the optimal heterogeneous single-ISA design by as much as 20.8% and improves energy efficiency over the most efficient single-ISA design by 23%.

Additionally, this paper builds on prior work in multi-ISA compilation by reducing compiled-code overhead to virtually zero, and by greatly decreasing the average distance to an equivalence point (which drives the average cost for binary translation, and thus migration).

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported in part by NSF Grants CCF-1219059 and CCF-1302682.

References

[1] 2nd Generation Intel Core vPro Processor Family. Technical report, Intel, 2008.

[2] The future is fusion: The Industry-Changing Impact of Accelerated Computing. Technical report, AMD, 2008.

[3] The Benefits of Multiple CPU Cores in Mobile Devices. Technical report, NVidia, 2010.

[4] Variable SMP – A Multi-Core CPU Architecture for Low Power and High Performance. Technical report, NVidia, 2011.

[5] ARM Limited. *ARM7TDMI Technical Reference Manual*.

[6] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Technical Conference*, Apr. 2005.

[7] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *Micro, IEEE*, 2006.

[8] E. Blem, J. Menon, and K. Sankaralingam. A Detailed Analysis of Contemporary ARM and x86 Architectures. Technical report, University of Wisconsin - Madison, 2013.

[9] E. Blem, J. Menon, and K. Sankaralingam. Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures. In *International Symposium on High Performance Computer Architecture*, Feb. 2013.

[10] N. Chitlur, G. Srinivasa, S. Hahn, P. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, et al. QuickIA: Exploring Heterogeneous Architectures on Real Prototypes. In *International Symposium on High Performance Computer Architecture*, Feb. 2012.

[11] M. DeVuyst, A. Venkat, and D. M. Tullsen. Execution Migration in a Heterogeneous-ISA Chip Multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012.

[12] Digital Equipment Corporation. *Alpha Architecture Reference Manual*.

[13] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *International Symposium on Computer Architecture*, June 2011.

[14] A. Ferrari, S. J. Chapin, and A. Grimshaw. Heterogeneous Process State Capture and Recovery through Process Introspection. *Cluster Computing*, 2000.

[15] P. Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. Technical report, ARM, 2011.

[16] M. Hill and M. Marty. Amdahl's Law in the Multicore Era. *Computer*, July 2008.

[17] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*.

[18] C. Isen, L. K. John, and E. John. A Tale of Two Processors: Revisiting the RISC-CISC Debate. In *Computer Performance Evaluation and Benchmarking*. 2009.

[19] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, July 2005.

[20] R. E. Kessler. The Alpha 21264 Microprocessor. *Micro, IEEE*, 1999.

[21] A. Krishnaswamy and R. Gupta. Efficient Use of Invisible Registers in Thumb Code. In *International Symposium on Microarchitecture*, Dec. 2005.

[22] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *International Symposium on Microarchitecture*, Dec. 2003.

[23] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2006.

[24] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *International Symposium on Computer Architecture*, June 2004.

[25] C. Lattner. LLVM and Clang: Next Generation Compiler Technology. In *The BSD Conference*, May 2008.

[26] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization*, Mar. 2004.

[27] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture*, Dec. 2009.

[28] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating System Support for Overlapping-ISA Heterogeneous Multi-Core Architectures. In *International Symposium on High Performance Computer Architecture*, Jan. 2010.

[29] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. In *ACM SIGMETRICS Performance Evaluation Review*, June 2003.

[30] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[31] P. Smith and N. C. Hutchinson. Heterogeneous Process Migration: The Tui System. *Software-Practice and Experience*, 1998.

[32] L. Strozek and D. Brooks. Energy-and Area-Efficient Architectures through Application Clustering and Architectural Heterogeneity. *ACM Transactions on Architecture and Code Optimization*, 2009.

[33] S. Terpe. Why Instruction Sets No Longer Matter. 2011.

[34] Texas Instruments Inc. *OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide*.

[35] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (pie). In *International Symposium on Computer Architecture*, June 2012.

[36] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010.

[37] D. G. Von Bank, C. M. Shub, and R. W. Sebesta. A Unified Model of Pointwise Equivalence of Procedural Computations. *ACM Transactions on Programming Languages and Systems*, 1994.

[38] V. M. Weaver and S. A. McKee. Code Density Concerns for New Architectures. In *International Conference on Computer Design*, Oct. 2009.