Composite-ISA Cores: Enabling Multi-ISA Heterogeneity Using a Single ISA

ABSTRACT

Heterogeneous multicore architectures comprise of multiple cores of different sizes, organizations, and capabilities. These architectures maximize both performance and energy efficiency by allowing applications to adapt to phase changes by migrating execution to the most efficient core. Multi-ISA heterogeneous architectures further take advantage of the inherent ISA-preferences of different code regions within an application to provide significant additional performance and efficiency gains.

This work proposes composite-ISA cores that implement composite feature sets made available from a larger superset ISA. This architecture has the potential to recreate and in many cases, supersedes the gains of multi-ISA heterogeneity, by leveraging a single composite-ISA. Composite-ISA CMPs enhance existing performance gains due to hardware heterogeneity by an average of 19%, and have the potential to achieve an additional 31% energy savings and 35% reduction in Energy Delay Product, without any loss in performance.

1. INTRODUCTION

Prior work has demonstrated the advantages of heterogeneous multicore architectures [1, 2, 3, 4, 5, 6] and in particular, multi-ISA heterogeneous multicore architectures [7, 8, 9, 10]. The advantages of the latter include greater diversity in design options leading to more efficient overall designs, enabled by the ability to more precisely cater each core to the specific needs of a subset of the code regions that will execute on the processor as a whole. The advantages of varying the ISA is that we can tune not just hardware features (cache sizes, window sizes, etc.), but can also tune features such as virtual register size, vector support, addressing mode availability, etc. to the needs of the code executing.

However, in prior work the ISA-dependent features had to be selected at a very coarse level. For example, you either take ISA 1 (e.g., x86 perhaps with one key feature you need but a lot of other baggage) or ISA 2 (e.g., Thumb which foregoes several features you don’t want but also a few you do). Such coarse selection greatly restricts both the ability to assign threads to the best core, but even more so it greatly restricts the processor architect in identifying the best combination of hardware features to assign to a globally optimal set of heterogeneous cores.

Additionally, heterogeneous-ISA architectures incur particular overheads, including fat binaries (which impact memory footprint but not cache footprints) and thread migration costs that include binary translation and stack transformation [7, 8]. But even those costs can pale compared to licensing, legal, testing, and verification costs and barriers when seeking to combine existing commercial ISAs on one chip [10, 11, 12].

In contrast, this work seeks to achieve (and extend) the benefits of multi-ISA heterogeneous designs while retaining the advantages of a single ISA – cheap migrations, minimal if any transformation, a single binary, and no binary translation. At the same time, we want to give designer and the code generator far more control over the choice of ISA, with the ability to make fine-grain choices about the features they need. By combining the ISA-affinity advantages of multi-ISA heterogeneous CMPs and the simplicity of single-ISA heterogeneous CMPs, we bring the best of both worlds in the composite-ISA heterogeneous architectures we propose.

We do this by creating a set of potential custom ISA designs which are all derived from a superset ISA. As a result, we maintain binary compatibility – any code region can run on any core that supports the subset of the ISA it was compiled to, or even with less efficiency on cores that don’t, given the high similarity in the ISAs. Thus, for each code region, the compiler chooses the set of features minimally needed, and then that region has the option of running on a smaller, more efficient core that does not need to support the full superset ISA. For this work, we construct the superset ISA using extensions and mechanisms completely consistent and compatible with the existing Intel x86 ISA, but even more could be done by creating a new ISA from scratch. We start with x86 both because it starts with a large set of the features we want, but also because it has a clear history and process for adding extensions. In particular we add more registers, more orthogonal selection of register width, and support for full predication.

We allow ISA customization along five axes: register depth (virtual registers supported), register width, instruction complexity, predication, and vector support. Ignoring some permutations of these features that do not appear to make sense, this still gives the processor designer 26 subset ISAs that are potentially useful. Thus, the process of designing an optimal set of core designs for a given expected workload mix is presented with far more flexibility. This work also features a comprehensive analysis of the hardware implications of these ISA feature set options, including a full synthesized RTL design of multiple versions of the x86 decoder.

This paper further describes our compiler infrastructure, which has the ability to make good choices for the features utilized by each code region, as well as ensure that the code is fully optimized by the compiler no matter what combination of features are selected. We describe the processor global design and optimization process, and show how the extra degree of flexibility enhances the capabilities of the design.

Due to the constraint of a single ISA, this approach can never match all advantages of distinct ISAs (such as the code compaction of Thumb). However, the greater flexibility and composable of our design give it other advantages. In area-constrained designs optimized for multithreaded performance, we gain 19% in performance and 35% in energy-delay product over single-ISA heterogeneous designs, essentially with a
While most compiler intermediate representations, including wider data types implies wider pointers allowing access to wider register depth to be customized to 8, 16, 32, and (e.g. dynamically scheduled), power and area will scale with spills, 10.3% in loads (refills), 3.5% increase in integer value to avoid spills/refills) [20, 21, 22, 23, 24]. For example, the underlying ISA’s register width not only requires more register depth. However, the physical register size scales with a core with greater register depth, and at the same time the register pressure of impacted code regions by migration customizes each core with a different register depth alleviates order buffer and the physical register file. In our superset ISA, it resembles x86, but with an additional set of features that can be customized along 5 different dimensions: register depth, register width, opcode and addressing mode complexity, predication, and data-parallel execution.

**Register Depth.** The number of programmable registers exposed by the ISA to the compiler/assembly programmer constitutes an ISA’s register depth. The importance of register depth as an ISA feature is well established due to its close correlation to the actual register pressure (number of registers available for use) in any given code region [13, 14, 15, 16]. While most compiler intermediate representations, including GCC’s Register Transfer Language [17, 18] and LLVM’s bytecode [19], allow for a large number (potentially infinite) of virtual registers, the number of architectural registers is limited, resulting in spills and refills of temporary values into memory.

Register depth not only affects efficient code generation, but significantly impacts machine-independent code optimizations due to (register pressure sensitive) redundancy elimination and re-materialization (re-computation of a temporary value to avoid spills/refills) [20, 21, 22, 23, 24]. For example, decreasing the register depth from 32 to 16 registers in our custom feature sets results in a increase of 3.7% in stores (spills), 10.3% in loads (refills), 3.5% increase in integer instructions and 2.7% in branch instructions (rematerialization) on the SPEC CPU2006 benchmarks compiled using the methodology described in Section 4.

Furthermore, the backend area and power is strongly correlated to the ISA’s register depth potentially impacting the nature and size of microarchitectural structures such as the reorder buffer and the physical register file. In our superset ISA, we allow register depth to be customized to 8, 16, 32, and 64 registers. A fully-custom heterogeneous multicore that customizes each core with a different register depth alleviates the register pressure of impacted code regions by migration to a core with greater register depth, and at the same time saves power by enabling smaller microarchitectural structures in other cores. In processors that support register renaming (e.g. dynamically scheduled), power and area will scale with physical register size rather than the ISA-supported virtual register depth. However, the physical register size scales with the virtual register depth.

**Register Width.** Like register depth, the register width of an architecture impacts performance and efficiency in several different ways. First, emulating data types wider than the underlying ISA’s register width not only requires more dynamic instructions, but could potentially use up more registers and thereby adversely impact register pressure. Second, wider data types implies wider pointers allowing access to larger virtual memory and avoiding unnecessary memory mapping to files. However, wide pointers potentially expand the cache working set when stored in memory as part of an aggregate object, thereby negatively impacting performance on applications that otherwise have smaller working sets [25]. Third, wider registers can often be addressed as individual sub-registers enhancing the overall register depth of the ISA. Most compilers’ register allocators take advantage of sub-registers and perform aggressive live range splitting and sub-register coalescing [23, 21, 26, 27, 28, 19].

Finally, wider registers imply larger architectural and physical register files in the processor which impacts both processor die area and overall power consumption. In our experiments, doubling the register file from 32-bits to 64-bits could impact processor power by as much as 6.4% across different register depth organizations. Our superset ISA supports both 32-bit and 64-bit wide registers like x86, but modify the instruction encoding to avoid any restrictions on the addressing of a particular register, sub-register, or a combination thereof, in any given instruction.

**Instruction Complexity.** The variety of opcodes and addressing modes offered by an instruction set controls the mix of dynamic instructions that enter and flow through the pipeline. Incorporating a reduced set of opcodes and addressing modes into the instruction set could significantly simplify the instruction decode engine, if chosen carefully. In particular, we strive for 1:1 instead of a 1:n macro-op to micro-op translation, saving as much as 9.8% in peak power and 15.1% in area. However, for some code regions, such a scheme could increase the overall code size potentially impacting both instruction fetch energy and instruction cache hit rate.

To derive such a reduced feature set, we carve out a subset of opcodes and addressing modes from our superset ISA that can be implemented using a single micro-op, essentially creating custom cores that implement the x86 micro-op ISA, albeit with variable-length encoding. The reduced feature set, called microx86 in this work, adheres to the load-compute-store philosophy followed by most RISC architectures. As a result, we could view this option as RISC vs CISC support. While one could conceive a more aggressive low-power RISC implementation of microx86 that implements fewer opcodes and further recycles opcodes for a more compact representation [29, 30], we follow x86’s existing variable-length encoding and 2-phase decoding scheme. This not only maintains consistency with existing implementations of x86, but prevents us from incurring the multi-ISA compilation and binary translation costs associated with fully heterogeneous-ISA designs. However, this does mean we cannot completely replicate the instruction memory footprint of a minimal RISC representation.

**Predication.** Predication converts control dependences into data dependences in order to eliminate branches from the instruction stream and consequently take pressure off the branch predictor and associated structures [31, 32, 33, 34, 35]. Modern ISA implementations of predication can be classified into three categories: (a) partial predication that allows only a subset of the ISA’s instructions to be predicated, (b) full predication that allows any instruction to be predicated using a predefined set of predicated registers, and (c) conditional execution that allows any instruction to be predicated using one condition code register.

The x86 ISA already implements partial predication via...
conditional move instructions that are predicated on condition codes. In this work, we add full predication support to our superset ISA, allowing any instruction to be predicated using any available general-purpose register using the if-conversion strategy described in Section 4. While predication eliminates branch dependences, aggressive if-conversion typically increases the number of dynamic instructions, thus placing more pressure on the instruction fetch unit and the instruction queue. In our custom feature sets that offer predication, we observe an average increase of 0.6% in the number of dynamic instructions with a reduction of 6.5% in branches.

Data-Parallel Execution. Most modern instruction sets offer primitives to perform SIMD operations [36, 37, 25, 38] to take advantage of the inherent data-level parallelism in specific code regions. The x86 ISA already supports multiple feature sets that implement a variety of SIMD operations. We include the SSE2 feature set in our superset ISA that can compute on data types that are as wide as 128 bits as implemented in the gem5 simulator [39]. Furthermore, we constrain our microx86 implementations to not include the SSE2 feature set since more than 50% of SIMD operations rely on 1:n encoding of macro-op to micro-op. In our fully-custom heterogeneous design, cores that do not implement SSE2 save 7.4% in peak power and 17.3% in area. They execute a precompiled scalarized version of the code when available, and in most cases, migrate code regions that enter intense vector activity to cores with full vector support.

As shown in Figure 1, we derive 26 different custom feature sets along the five dimensions described above. Note that we exclude predication from our 32-bit feature sets that have less than 16 registers since they already suffer from extremely high register pressure. Similarly, we constrain 64-bit ISAs to have a register depth that is greater than or equal to 16.

Figure 2 shows the instruction (micro-op) breakdown for the SPEC CPU2006 benchmarks on three different custom ISAs: (a) the 32-bit version of microx86 with a register depth of 8 and with no additional features (smallest feature set in our exploration), (b) the x86-64 ISA with SSE and no other customizations, and (c) the superset ISA which implements all the features described above. Due to the high register pressure in microx86-32, it incurs an average of 28% higher memory references than x86-64 and an overall expansion of 11% in the number of dynamic instructions. Also compared to the x86-64, we find that the superset ISA, owing to the diverse set of custom features added, sees an average reduction of 8.5% in loads (spill elimination), 6.3% in integer instructions (aggressive redundancy elimination), and 3.2% in branches (predication).

3. DECODER DESIGN AND IMPLEMENTATION

In this section, we describe our customizations to the x86 instruction encoding and decoder implementation in order to support the 26 feature sets derived in the previous section.

3.1 Instruction Encoding

Feature extensions to the x86 instruction set is not uncommon. In accordance to its code density and backward compatibility goals, major feature set additions to x86 (e.g., REX, VEX, and EVEX) have been encoded by exploiting unused opcodes and/or by the addition of new (optional) prefix bytes. We use similar mechanisms to encode the specific customizations we propose in this work. Figure 3 shows our customizations in detail. In order to double/quadruple the register depth of x86-64,
we add a new prefix – REXBC, similar to the addition of
the REX (register extension) prefix that doubled both the
register width and depth of x86, giving rise to the x86-64
ISA. In particular, the REXBC prefix encodes 2 extra bits for
each of the 3 register operands (input/output register, base,
and index), which is further combined with 4 bits from the
REX, MODRM, and SIB bytes, to address any of the 64
programmable registers. Furthermore, we use the remaining
2 bits of the REXBC prefix to lift restrictions in the x86
and x86-64 ISAs that do not allow certain combinations of
registers and subregisters to be used as operands in the same
instruction. Finally, we exploit an unused opcode 0xd6
in order to mark the beginning of a REXBC prefix.

Similarly, to support predication, we use a combination of
an unused opcode 0xf1 and a predicate prefix. In order to
efficiently support diamond predicate (described in Sec-
4), we use the predicate prefix to encode both the nature
(positive/negative) of the predicate (bit 7) and the register
(bits 0-6) the instruction is predicated on.

All of the insights in this paper apply equally (if not more
so) to a new superset ISA designed from scratch for this
purpose – such an ISA would allow much tighter encoding
of these options.

3.2 Decoder Analysis

Figure 4 shows the step-by-step decoding process of a typ-
ical x86 instruction. Owing to the variable length encoding,
x86 instructions typically go through a 2-phase decode pro-
cess. In the first phase, an instruction-length decoder fetches
and decodes raw bytes from a prefetch buffer, performs length
validation, and further queues the decoded macro-ops into
a macro-op buffer. These macro-ops are fused into a single
macro-op when viable, and further fed as input into one of the
instruction decoders that decode it into one or more micro-
ops. The decoded micro-ops are subjected to fusion again,
in order to store them in the micro-op queue and the micro-
op cache in a compact representation. The micro-op cache
is both a performance and power optimization that allows
the decode engine to stream decoded (and potentially fused)
 micro-ops directly from the micro-op cache, turning off the
rest of the decode pipeline until there is a micro-op miss.

In our RTL implementation, we model an instruction length
decoder based on the parallel instruction length decoder
described by Madduri et al [40]. The instruction length decoder
has three components: (a) an instruction decoder that de-
codes each byte of an incoming 8-byte chunk as the start
of an instruction, decoding both prefixes and opcodes, (b)
a speculative length calculator that speculatively computes
the length of the instruction based on the decoded prefixes
and opcodes, and (c) an instruction marker that checks the
validity of the computed lengths, marks the begin and end of
an instruction, and detects overflows into the next chunk.

Since our customizations affect the prefix part of the in-
struction, we modify the eight decode subunits of the in-
struction decoder to include comparators that generate extra
decode signals to represent the custom register depth and
predicate prefixes. Furthermore, these decode signals prop-
agate through the speculative instruction length calculator
and the instruction marker requiring wider multiplexers in
the eight length subunits, the length control select unit, and
the valid begin unit. These modifications to the instruction
length decoder result in an increase of 0.87% in total peak
power and 0.65% in area for our superset ISA.

Furthermore, we increase the width of the macro-op queue
by 2 bytes to account for the extra prefixes. Since predication
support and greater register depth in our superset ISA could
potentially require wider micro-op ISA encoding, we increase
the width of the micro-op cache and the micro-op queue by 2
bytes. Finally, for our microx86 implementations, we replace
the complex 1:4 decoder with a simple 1:1 decoder and forgo
the microsequencing ROM. From our analysis, a decoder that
implements our simplest feature set microx86-32 consumes
0.66% less peak power and takes up 1.12% lesser area than
the x86-64 decoder, and our superset decoder consumes 0.3%
more peak power and takes up 0.46% more area than the
x86-64 decoder. These variances do not include the increases
or savings from the instruction length decoder.

4. COMPILER AND RUNTIME STRATEGY

One of the major goals of this paper is to provide the
benefits of ISA diversity without incurring the multi-ISA
compilation, transformation, and binary translation costs, by
leveraging custom feature sets of the same ISA. In this sec-
tion, we describe our compilation strategy that generates
code to efficiently take advantage of the underlying custom
feature sets, and our runtime migration strategy that allows
code regions to rapidly migrate back and forth between dif-
ferent custom feature sets, without the overhead of full binary
translation and/or stack transformation.

4.1 Compiler Toolchain Development

Compilation to a superset ISA or a combination of custom
feature sets allows different code regions to take advantage
of the variety of custom feature sets implemented by the
underlying hardware. For example, code regions with high
register pressure could be compiled to execute on a feature
set with greater register depth, and code regions with too
many branches could be compiled to execute predicated code.
We leverage the LLVM MC infrastructure [19] to efficiently
encode the right set of features supported by the underlying
custom design and further propagate it through various in-
struction selection, code generation, and machine-dependent
optimization passes. We further take advantage of the MC
code emitter framework to encode feature sets such as register
depth and predication that require an extra prefix.

In order to convert the existing x86 backend to that of the
superset ISA, we first include the additional 48 registers in
the ISA’s target description and further associate code density
 costs with it. This enables the register allocator to always
prioritize the allocation of a register that requires fewer prefix
bits to encode it. Furthermore, we allow each of these regis-
ters to be addressed as a byte, a word, a doubleword, and a
quadword register, thereby smoothly blending into the regis-
ter pressure tracking and subregister coalescing framework.

We next implement full predication in x86 by re-purposing
the existing machine-dependent if-conversion framework of
LLVM that implements if-conversions in three scenarios:
(a) diamond – when a true basic block and a false basic
block split from an entry block and rejoin at the tail, (b)
triangle – when the true block falls through into the false
block, and (c) simple – when the basic blocks split but do not
rejoin, such as a break statement within a conditional. For
every such pattern in the control flow graph, if-conversion
is performed if determined profitable. The profitability of

4
if-conversion is based on branch probability analysis, the approximate latency of machine instructions that occur in each path, and the configured pipeline depth. We further add the if-conversion pass as a pre-scheduling pass for the X86 target—this allows the scheduling pass to take full advantage of the large blocks of unscheduled code created by the if-conversion.

To implement the microx86 feature set, we modify the existing x86 backend to exclude all opcodes, addressing modes, and prefixes that decode into more than one micro-op from the machine instruction selection process. While LLVM’s instruction selector, for the most part, identifies a replacement ld-compute-st combination for the excluded addressing mode, certain IR instructions such as tail jumps and tail call returns require explicit instruction lowering.

### 4.2 Migration Strategy

Process migration across overlapping custom feature sets could involve two scenarios. In an upgrade scenario, a process is compiled to use only a subset of the features implemented by the core to which it is migrated, and therefore can resume native execution immediately after migration. Conversely, in a downgrade scenario, the core to which a process is migrated implements only a subset of the features being used by the process, which necessitates binary translation and/or program state transformation. Consequently, we identify the following schemes of migration.

**Free Upgrades and Compiler-directed Downgrades** In this scheme, the compiler analyzes specific basic blocks in the program and marks them as downgrade-safe to a particular feature set, if there is no other reachable basic block that implements features outside the given feature set. Such a migration scheme works well in case of single-threaded workloads where there is no competition for resources (including feature sets) and effective compilation ensures that a thread is always allocated the optimal set of resources—i.e., the code always runs on the feature set it was compiled for.

**Free Upgrades and Emulated Downgrades**. While the compiler-directed downgrades works well in certain scenarios, static scheduling does not scale with multiple workloads, changing input sizes, and changing environmental constraints. For example, a thread may run on a core it was not compiled for if another is already scheduled there. Additionally, you may have code that should take full advantage of 64 registers when the battery is full, but would be better with 16 registers when the battery is low.

Therefore, we propose another migration strategy that performs feature emulation (via dynamic binary translation) in the event of a downgrade. Owing to the overlapping nature of the feature sets (same opcode and instruction format), feature emulation involves only a small set of binary code transformations. First, when we downgrade from x86 to microx86, we perform simple addressing mode transformations by converting any instruction that directly operates on memory into the ld-compute-st format. Second, we downgrade to a feature set with a smaller register depth by translating higher registers into memory operands using a register context block [41, 7], a commonly used technique during binary translation to register-constrained ISAs. Third, we perform long-mode emulation [8] and use fat pointers implemented using xmm registers in order to emulate wider types on a 32-bit core. Finally, we employ reverse if-conversions to translate predicated code back to control-dependences.

**Fat-binary.** Finally, we also evaluate the multi-ISA compiler-assisted execution migration strategy, primarily to compare against prior work [7, 8, 9, 42, ?] that migrate execution across fully heterogeneous ISAs. This involves construction of a fat binary with multiple feature set-dependent code sections and a single copy of feature set-independent data sections. Furthermore, at the time of migration, we perform feature set emulation for a brief period of time until we reach a compiler-marked equivalence point. We transform the program state using hints from the multi-feature-set compilation, at which point of time, the downgrade is complete.

### 5. METHODOLOGY

In this section, we describe our experimental methodology for the feature set exploration and process migration strategy. As shown in Table 1, our design space consists of 5 dimensions of ISA customizations and 19 micro-architectural dimensions, resulting in 26 different custom feature sets, 180 microarchitectural configurations, and 4680 distinct single core design points, that each are spread across a wide range of peak power (4.8W to 23.4W) and area (9.4mm² to 28.6mm²) distributions. The goal of our feature set exploration is to find an optimal 4-core multicore configuration using the fully custom feature sets derived out of the superset ISA. Our objective functions that evaluate optimality include both performance and energy delay product (EDP), for both multithreaded and single-threaded workloads.

Our workloads include eight SPEC CPU2006 benchmarks further broken down into 49 different application phases using the SimPoint [43, 44] methodology. In order to create equivalent phases for 26 different feature sets across the 8 applications, we first create simpoints on the commonly used x86-32 ISA with a simpoint interval of 100 million dynamic instructions, and then find equivalent start and end basic blocks using a combination of IR-level and MC-level data obtained using LLVM, and simpoint metadata obtained using the fast (atomic) simulation in the gem5 architectural simulator [39]. Our design space exploration selects opti-

<table>
<thead>
<tr>
<th>ISA Parameter</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register depth</td>
<td>8, 16, 32, 64 registers</td>
</tr>
<tr>
<td>Register width</td>
<td>32-bit, 64-bit registers</td>
</tr>
<tr>
<td>Instruction/Addressing</td>
<td>1:1 macro-op/micro-op encoding</td>
</tr>
<tr>
<td>mode complexity</td>
<td>(load-store x86 micro-op ISA), (fully CISC x86 ISA)</td>
</tr>
<tr>
<td>Instruction/Addressing</td>
<td>1:1 macro-op/micro-op encoding</td>
</tr>
<tr>
<td>mode complexity</td>
<td>(load-store x86 micro-op ISA), (fully CISC x86 ISA)</td>
</tr>
<tr>
<td>Predication Support</td>
<td>Full Predication like IA-64/Hexagon</td>
</tr>
<tr>
<td>Data Parallelism</td>
<td>Scalar vs Vector (SIMD) execution</td>
</tr>
<tr>
<td>Microarchitectural Parameter</td>
<td>Options</td>
</tr>
<tr>
<td>Execution Semantics</td>
<td>Inorder vs Out-Of-Order designs</td>
</tr>
<tr>
<td>Fetch/Issue Width</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td>Decoder Configurations</td>
<td>1:3 1:1 decoders, 1:1:1 decoder, MSROM</td>
</tr>
<tr>
<td>Micro-op Optimizations</td>
<td>Micro-op Cache, Micro-op Fusion</td>
</tr>
<tr>
<td>Instruction Queue Sizes</td>
<td>32, 64</td>
</tr>
<tr>
<td>Reorder Buffer Sizes</td>
<td>64, 128</td>
</tr>
<tr>
<td>Physical Register File</td>
<td>(64 INT, 64 FP/SIMD), (64 INT, 96 FP/SIMD)</td>
</tr>
<tr>
<td>Configurations</td>
<td></td>
</tr>
<tr>
<td>Branch Predictors</td>
<td>2-level local, gshare, tournament</td>
</tr>
<tr>
<td>Integer ALUs</td>
<td>1, 3, 6</td>
</tr>
<tr>
<td>FP/SIMD ALUs</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td>Load/Store Queue Sizes</td>
<td>16, 32</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>32KB 4-way, 64KB 4-way</td>
</tr>
<tr>
<td>Private Data Cache</td>
<td>32KB 4-way, 64KB 4-way</td>
</tr>
<tr>
<td>Shared Last Level (L2) Cache</td>
<td>4-banked 4MB 4-way, 4-banked 8MB 8-way</td>
</tr>
</tbody>
</table>

Table 1: Feature Set Exploration Space
mal multicore configurations by scheduling over all possible permutations of the workloads.

We use the gem5 simulator to measure performance in both our inorder and out-of-order cores. We modify the gem5 simulator to include a micro-op cache and micro-op fusion in order to measure the feature set impact in light of existing micro-op optimizations. However, we do not employ micro-op fusion in our microx86 ISA because each instruction only decomposes into one micro-op and the micro-op fusion unit doesn’t yet combine micro-ops from different macro-ops. Our implementations of the micro-op cache and micro-op fusion are consistent with guidelines mentioned in the Intel Architecture Optimization Manual [?].

We perform a full RTL synthesis using the Synopsis Design Compiler to measure the decoder area and power overheads of each of the customizations we employ in our feature sets, as described in Section 3. We also use the McPAT [45] power modeling framework to evaluate the power and area of the rest of the pipeline. The peak power and area measurements we obtain out of McPAT simulations are key parameters to our design space navigation.

We note that our design space exploration involves a massive effort of 196,560 gem5+McPAT simulations resulting in 49733 core hours on a large 2 petaflop cluster. Furthermore, our multicore design search involves finding an optimal 4-core multicore out of a 102.5 trillion combinations that run all permutations of simpoint regions, for several different objective functions and constraints. As of this writing, the results we report for the fully custom feature set design are local optima, and therefore conservative estimates.

Finally, to evaluate the efficiency of our migration strategy, we perform feature set emulation, compiler-directed feature set downgrades, and program state transformation-based process migration on each of the checkpointed regions that represent our simpoints. We report overall execution time overheads for different types of feature set emulations and full binary translation, distances to downgrade-safe points, and overall performance boost due to migration.

6. RESULTS

In this section, we present detailed results from our design space exploration that seeks to identify an optimal combination of ISA and microarchitectural features that supersedes hardware heterogeneity. We isolate specific ISA features that improve single thread performance and/or increase multi-programmed workload throughput/efficiency, and further study their effect on important architectural design choices that enable efficient transistor investment on the available silicon real estate.

6.1 Performance and Energy Efficiency

This section elaborates on the findings of our design space exploration. We identify custom multicore designs that benefit from both hardware heterogeneity and feature set diversity, providing significant gains over designs that exploit only hardware heterogeneity. Furthermore, these designs recreate the effects or in most cases, surpass the gains offered by a fully heterogeneous-ISA CMP that implements a completely disjoint set of vendor-specific ISAs and requires sophisticated OS/runtime support. We conduct multiple searches through our design space to model different execution scenarios and different budget-constrained environments.

<table>
<thead>
<tr>
<th>microx86-8D-32W</th>
<th>microx86-32D-64W</th>
<th>x86-16D-64W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thumb-like Features</td>
<td>Alpha-like Features</td>
<td>x86-64-like Features</td>
</tr>
<tr>
<td>Load/Store Architecture</td>
<td>Register Depth: 32</td>
<td>SIMD Support</td>
</tr>
<tr>
<td>Register Depth: 8</td>
<td>Register Width: 64</td>
<td>CISC Architecture</td>
</tr>
<tr>
<td>No SIMD support</td>
<td>No SIMD support</td>
<td>Register Width: 64</td>
</tr>
<tr>
<td>Exclusive Features:</td>
<td>Exclusive Features:</td>
<td>SIMD support</td>
</tr>
<tr>
<td>FP Support</td>
<td>None</td>
<td>CMOV Support</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>microx86-8D-32W</th>
<th>microx86-32D-64W</th>
<th>x86-16D-64W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thumb-specific Features:</td>
<td>Alpha-specific Features:</td>
<td>x86-specific Features:</td>
</tr>
<tr>
<td>Code Compression</td>
<td>Fixed-length instructions</td>
<td>None</td>
</tr>
<tr>
<td>(one-step decoding)</td>
<td>Fixed-length instructions</td>
<td>More FP Registers</td>
</tr>
<tr>
<td>(one-step decoding)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: x86-ized versions of Thumb, Alpha, and x86-64

In each search, we identify three optimal 4-core multicore designs: (1) homogeneous x86-64 CMPS that employ cores that implement the same ISA and the same microarchitecture, (2) x86-64 CMPS that exploit hardware heterogeneity alone, and (3) fully custom x86-64 CMPS that exploit hardware heterogeneity and full feature diversity.

In addition, we also identify two intermediate design points of interest: (1) fully heterogeneous-ISA CMPS that implement three fixed, disjoint, vendor-specific ISAs (x86-64, Alpha, Thumb), and (2) x86-64 CMPS that exploit hardware heterogeneity and a limited form of feature diversity via three x86-based fixed feature sets that resemble the above vendor-specific ISAs. We use the latter design as a vehicle to demonstrate that the vendor-specific ISA-heterogeneity offered by the former design can be recreated to a large extent by carving out custom feature sets from a single sufficiently-diverse superset ISA. Table 2 offers a more detailed comparison.

Thus, we compare against a number of interesting configurations, but two are most revealing. Since our goal is to seek to replicate the advantage of multi-ISA heterogeneity over single-ISA heterogeneity, the single-ISA heterogeneous result is our primary baseline for comparison. However, the multi-vendor (x86, Thumb, Alpha) result represents our “goal” result that we are striving to match, yet with essentially a single ISA.

Figure 5 compares the performance and energy efficiency of the five optimal designs listed above, optimized to provide multi-programmed workload throughput when constrained under different peak power and area budgets. There are four major takeaways from this experiment. First, the designs that exploit feature diversity alongside hardware heterogeneity consistently and significantly outperform the designs that exploit only hardware heterogeneity. This is because hardware heterogeneity tends to diminish when the amount of available chip real estate becomes too small or too generous. Second, we find in the resulting optimal architectures that in an especially tightly power/area constrained environment, every feature present in the superset ISA is implemented by at least one core in the fully custom design. When the constraints become more relaxed, the fully custom designs continue to implement at least 10 out of the 12 features described in Section 2. Third, the custom designs that implement the x86-ized versions of the vendor-specific ISAs trail slightly but generally match the gains of the fully heterogeneous-ISA designs. Fourth, the custom design with full ISA feature set diversity not only matches but frequently outperforms the true heterogeneous-ISA design. This indicates that any loss due to the lack of specific ISA encoding and simplified hard-
we observe that every feature of the superset ISA again man-
ware (e.g., decoders) is more than compensated for by the
increase flexibility of the composable ISA features.
Overall, the x86-64 designs that exploit both hardware
heterogeneity and full feature diversity outperform those that
only exploit hardware heterogeneity by an average of 17.6%,
and as much as 30% in tightly area-constrained environments.
In Figure 6, we compare designs optimized to provide
multi-threaded workload energy efficiency, measured as En-
ergy Delay Product (EDP), while being constrained under
different peak power and area budgets. We observe signifi-
cant energy savings due to full ISA customization – an aver-
age of 31% savings in energy and 34.6% reduction in EDP
over single-ISA heterogeneous multicore designs. This result
may be unintuitive, as the Thumb architecture still provides
significant advantages over our most conservative microx86
core. However, many codes cannot use Thumb because of its
limited features, while the Composite-ISA architecture can
combine microx86 with a variety of other features and make
use of it far more often.
We next evaluate our designs optimized to provide high
single thread performance and energy efficiency. When the
designs are constrained by peak power budgets, we model
them after the dynamic multicore topology [46] where only
one core is active at any given point of time, while the rest of
the cores are powered off.
Figure 7 compares designs optimized to provide high
single thread performance and energy efficiency. Note that
the peak power budgets are tighter in this case since we as-
sume only one core to be powered on at a time. Due to the
low power constraints, hardware heterogeneity provides only
marginal improvements in performance and EDP. However,
we observe that every feature of the superset ISA again man-
ifests in at least one of the feature sets implemented by the
fully custom design, allowing applications to migrate across
different cores in order to take advantage of any required
feature set. We observe an average speedup of 19.5% and
an average EDP reduction of 27.8% over single-ISA hetero-
geneous designs. Moreover, owing to the many low-power
and feature-rich microx86 options available, we manage to
outperform the fully heterogeneous-ISA design that imple-
ments vendor-specific ISAs, by 14.6% and reduce EDP by
12% under tight (5W) power constraints. The x86-ized ver-
sions of the fully heterogeneous-ISA designs again trail, but
generally match up well, to the performance and EDP levels
offered by vendor-specific ISA-heterogeneity. Once again,
the composite-ISA results overall match in performance, and
clearly exceed in EDP, the true heterogeneous-ISA results.
When designs are constrained by area budget, the optimal
multicore is typically composed of multiple small cores and
one large core that maximizes single thread performance. In
Figure 8, we evaluate the single thread performance and en-
ergy efficiency of the optimal designs under different area
budgets. We observe that the fully custom designs sport
two out-of-order microx86 cores even under the most tightly
area-constrained environment implementing different register
depth and width features in each of them, allowing appli-
cations to migrate across cores and take advantage of the
specific feature sets. While the fully heterogeneous-ISA de-
sign offers similar capabilities due to the area-efficient thumb
cores, we note that migration across thumb and x86-64 cores
is non-trivial and incurs significant overhead in comparison to
the simpler migration across the two overlapping x86-based
ISAs in our case. Overall, we find that the fully custom design
consistently outperforms the single-ISA heterogeneous de-
sign, resulting in an average speedup of 20% and a reduc-
tion of 21% in EDP.
Owing to their feature-rich nature, fully custom heterogeneous CMPs consistently offer significant performance and energy efficiency benefits, even in scenarios when hardware heterogeneity provides diminishing returns. One of the major goals of this design space exploration is to identify specific ISA features that contribute toward these benefits, and further help architects make more efficient design choices. However, since ISA features typically manifest as components of a larger feature set a core implements, it is non-trivial to measure the effect of a specific ISA feature in isolation.

In this section, we perform additional searches through our design space in order to understand the impact an ISA feature has over performance, energy, and transistor investment, by removing one axis of feature diversity at a time. As an example, consider the search for an optimal fully custom CMP that optimizes for multi-programmed workload performance under an area budget of 48mm², but constrained to only include designs that limit the number of architectural registers to 16 in all cores. If register depth is an important feature, the most optimal design from this search is expected to perform worse than the one chosen through an unconstrained search.

Figure 9 shows the result of this experiment. We make the following observations. First, constraining all cores to implement fewer than 32 architectural registers negates a significant chunk of the performance gain due to feature diversity. Most optimal designs typically employ two or more cores with a register depth greater than or equal to 32, and seldom employ cores with fewer than 16 registers. Second, the best performing designs typically include a mix of both 32-bit and 64-bit cores. While 64-bit cores are more efficient at computing on wider data types, 32-bit cores employ smaller hardware structures, saving area for other features. Designs that exclude any one of them incur 3-7% loss in performance.
it is the only design among the 10 that employs all out-of-order cores, each sporting a tournament branch predictor. Second, the design constrained to exclude micro86 takes up the highest processor area, investing most of its transistors on functional units. Note that we always combine SIMD units with x86 cores, and that the micro86 cores lack any SIMD units. Third, the 64-bit-only optimal design spends more transistors on the register file and the scheduler than any other design. In that design, two out of four 64-bit cores are configured with a register depth of 64, with the remaining two configured with 32.

Figure 11 shows the processor energy breakdown by stages for each of the best designs from the above experiment. We find that the energy breakdown shows significant deviation from the corresponding area breakdown. While the decoder makes up for a greater portion of processor area than the fetch unit, it is the fetch unit that expends more energy during run-time since the decode pipeline is only triggered upon a micro-op cache miss, and instructions are streamed out of the micro-op cache for the most part. Interestingly, the design that constrains all cores to be configured with a register depth of 8 spends significant energy in the Fetch stage. This is due to the artificial instruction bloat caused by spills, refills, and rematerializations – a direct consequence of high register pressure. Furthermore, although the x86-only designs invested significantly in SIMD units, the energy spent by the functional units is not nearly as proportional. This is due to relatively infrequent vector activity. Finally, the 64-bit-only design continues to dominate in terms of register file and scheduler energy.

### 6.3 Feature Affinity

In this section, we study the feature affinity of the eight applications we benchmark, in two specific execution scenarios. In the first scenario, we consider a fully custom heterogeneous design optimized for single thread performance under a peak power budget of 10W. Recall that in such a design, hardware heterogeneity provides only marginal benefits, and most of the gains come from the fact that an application is free to migrate across different cores that each implement a diverse feature set. Therefore, such a design captures the true ISA affinity of an application.

Figure 12 shows the feature affinity in terms of the fraction of time an application spends executing on a particular feature set. First, we find that the multicore design that optimizes single thread performance exhibits significant feature diversity. In fact, by analyzing the component features of the multicore, we find that all features from our superset ISA have been used. Second, we find that there is significant variance in feature set preference across different applications. There is no single best feature set that is preferred by all applications. Third, we observe that there is some variance in feature set preference even within a single application’s execution phases. We find that most applications migrate to a core with a different feature set at least once. Fourth, the benchmark hammer, which exhibits significant register pressure tends to always execute on a feature set with a register depth of 64. Interestingly, we found that phases with considerable irregular branch activity, due to indirect branches and function pointer calls, prefer fully predicated instruction sets since they ease the pressure on the branch predictor by converting some of the control flow into data flow. This is evidenced by the benchmarks sjeng and gobmk.
In the second scenario, we consider a fully custom design optimized for multi-programmed workload throughput under an area budget of 48mm², in which case applications typically contend for the best feature set preference, and may sometimes execute on feature sets of second preference. Figure 13 shows the results of this experiment. In sharp contrast to the design optimized for single thread performance, where applications had clear feature set preferences, we find that all applications in the multi-programmed workload execute on all feature sets at some point of time. However, we are still able to make some high level inferences about feature affinity. For example, the benchmark sjeng continues to show a clear preference to x86 over microx86, and both benchmarks sjeng and gobmk prefer to execute on fully predicated ISAs during phases of irregular branch activity.

### 6.4 Migration Cost Analysis

Process migration across composite-ISA cores can involve two scenarios. In a downgrade scenario, the core to which the process migrates implements only a subset of the features the process is compiled to, necessitating software emulation of unimplemented features via binary translation and/or program state transformation. On the other hand, in an upgrade scenario, the core which a process migrates to already implements a superset of the features the process was compiled to, in which case, there is no migration cost in terms of program state transformation. While a profile-guided/compiler-directed static scheduling scheme could technically eliminate feature downgrade costs, we want to study a more dynamic scheme in which an application identifies the feature set of its preference and migrates to it at run-time. In this section, we evaluate the following dynamic migration schemes.

**Free upgrades and Emulated Downgrades.** This scheme works in the context of a runtime system that software emulates instructions that use unimplemented features. Figure 14 shows feature downgrade costs incurred due to the software emulation framework. We make several important observations here. First, when we downgrade from 64-bit to 32-bit cores, most of the emulation cost is negated due to the cache-efficient 32-bit cores. In fact, we achieve a speedup for some applications when we downgrade them from 64-bit to 32-bit feature sets. Second, since most applications use 32 or fewer registers, there is little emulation cost incurred due to a register depth downgrade from 64 to 32 registers. While we incur some overhead (an average of 2.7%) when we downgrade to a feature set that implements only 16 registers, there is significant overhead (an average of 33.5%) in migrating to a feature set that implements only 8 registers. In all cases, we find that the benchmark hmmer incurs the highest emulation overhead due to a register depth downgrade, concurring with our prior feature affinity analysis. Third, we incur an average of 5.5% overhead when we downgrade to a feature set without full predication. We note that this is highly dominated by the outlier libquantum, which in our best designs always executes on a partially predicated feature set since the compiler tends to overestimate the cost of diamond predication for this benchmark. Finally, downgrade from x86 to microx86 comes at a cost of 4.2% average. This can be attributed to the emulation of almost every arithmetic instruction that uses indirect addressing mode.
Fat Binary Approach. Although the fat binary approach that relies on the multi-ISA compilation strategy proposed by Devuyst, et al. [7] seems heavy-weight when applied to composite-ISA architectures that implement overlapping-ISA, we include it in our study since it offers an interesting comparison. Due to the overlapping nature of most feature sets, we manage to be migration-safe on an average of 64.7% of the basic blocks with many instances where we are migration-safe for more than 90% of the basic blocks. This is in comparison to prior work on multi-ISA architectures where only 48% of the basic blocks were transformable. Moreover, we tend to transform fewer objects on the stack due to common addressing modes.

Figure 15 offers a comparison of the two techniques in the context of designs optimized for multi-programmed workload throughput. Overall, the migration cost with the fat binary approach cause 0.43% performance degradation, in comparison with the emulated downgrades approach that limits performance degradation to just 0.06%, virtually preserving all of the performance gains due to feature diversity. We attribute such a small migration cost to the fact that most migrations do not require a downgrade and when there is a downgrade, the cost of emulating such a downgrade is minimal, both due to the overlapping nature of feature sets.

In summary, we see that composite-ISA heterogeneous designs consistently outperform and use far less energy than single-ISA heterogeneous designs.

8. CONCLUSION

This paper presents a composite-ISA architecture and compiler/runtime infrastructure that extends the advantages of multi-ISA heterogeneous architectures. It does so along two dimensions. First, it enables the full performance and energy benefits of multi-ISA design, without the issues of multi-vendor licensing, fat binaries, expensive migrations, stack transformations, etc. Second, it provides both the processor designer and the compiler a much richer set of ISA design choices, enabling them to select and combine features that match the expected workload. This provides richer gains in efficiency than highly optimized but inflexible existing-ISA based designs.

Under certain design scenarios, this architecture gains 30% in performance and over 30% in energy-delay product over single-ISA heterogeneous designs. Further, it matches and in many cases outperforms the full Heterogeneous-ISA (x86 + Alpha + Thumb) design results, and consistently runs at lower levels of energy-delay product.
9. REFERENCES


[38] Intel, Intel 64 and IA-32 Architectures Software Developer’s Manual.


