ISA-Independent Workload Characterization and its Implications for Specialized Architectures

Yakun Sophia Shao and David Brooks
Harvard University
{shao,dbrooks}@eecs.harvard.edu

***Abstract*—Specialized architectures will become increasingly important as the computing industry demands more energy-efficient designs. The application-centric design style for these architectures is heavily dependent on workload characterization of intrinsic program characteristics, but at the same time these architectures are likely to be decoupled from legacy ISAs. In this work, we perform ISA-independent workload characterization for a variety of important intrinsic program characteristics relating to computation, memory, and control flow. The analysis is performed using a JIT compiler that emits ISA-independent instructions. We compare this analysis with an x86 trace and find that several of the analyses are highly sensitive to the ISA. We conclude that designers of specialized architectures must adopt ISA-independent workload characterization approaches.**

I. Introduction

Specialized architectures are emerging as the major driving force for energy efficient design. Specialization seeks to harness characteristics of specific workloads, or categories of workloads, to enable more efficient computing hardware. Architectural specialization can take many forms. At one end of the spectrum are fully programmable, general purpose processing elements evolving from today's flexible, but inefficient, cores. At the other end of the spectrum are fixed-function accelerators providing large efficiency gains for very specific tasks such as video encoding, speech processing, or graph analysis. GPUs and other programmable data parallel architectures fit between these two extremes. Many of these architectural approaches are not tied to a specific legacy instruction set architecture (ISA), and in some such designs ISAs are eschewed completely.

Specialized architectures are intrinsically tailored to applications, and workload characterization will play a large role in developing these architectures. Tuning an architecture towards a workload requirement demands a comprehensive understanding of the intrinsic characteristics of the workload. Workload characterization for general-purpose architectures is commonly done by profiling benchmarks on current generation microprocessors using hardware performance counters. Typical program characteristics are machine instruction mix, IPC, cache miss rates, and branch misprediction rates. This approach is limited because machine-dependent features such as cache size and pipeline depth will strongly impact the workload characterization. To overcome the problem, *microarchitecture-independent* workload characterization can be employed by profiling instruction traces to collect information such as working set sizes, register traffic, memory locality, and branch predictability [11]. Although this approach removes the effects of microarchitecture-dependent features, some of these analyses depend on the particular ISA with which the trace is represented. Each ISA has different characteristics and constraints that impact the representation of the workload. As architectural specialization grows in importance, *ISA-independent* workload characterization will become essential for understanding intrinsic workload behavior, which will in turn allow designers to consider a wide range of alternative architectures.

To fully expose the microarchitecture- and ISA-independent workload characteristics for specialized architectures, we propose to analyze benchmarks using ISA-independent characteristics that capture inherent program behavior. In order to perform this analysis, we leverage the existing ISA-independent nature of a compiler intermediate representation (IR). We use a JIT compiler to trace workloads using this ISA-independent program representation and compare program characterization within the broad categories of program compute, memory activity, and control flow. In particular, we study program characteristics that are highly relevant to the design of specialized architectures. Within each category, we analyze and discuss the differences between ISA-independent and ISA-specific analysis. Finally, we demonstrate cases where the ISA-independent characterization can help designers categorize workloads into different specialization approaches. In particular, this paper makes the following contributions:

1) We compare ISA-dependent characterization with ISA-independent characterization. To the best of our knowledge, this is the first such ISA-independent workload characterization study. We show that ISA-dependent results can be misleading. In particular, the memory behavior of the workloads, which is critical for many forms of architectural specialization, will be biased significantly due to the register spilling effect intrinsic to conventional ISAs.
2) We present a taxonomy to characterize the potential for architectural specialization using ISA-independent characteristics. We categorize a workload's ISA-independent characteristics into program compute, memory activity, and control flow, each of which corresponds to an important component of specialized architectures.
3) We present workload characterization of SPEC CPU benchmarks using ISA-independent characteristics, and we demonstrate that a truly intrinsic workload characterization allows accelerator designers to quickly identify opportunities for specialization.

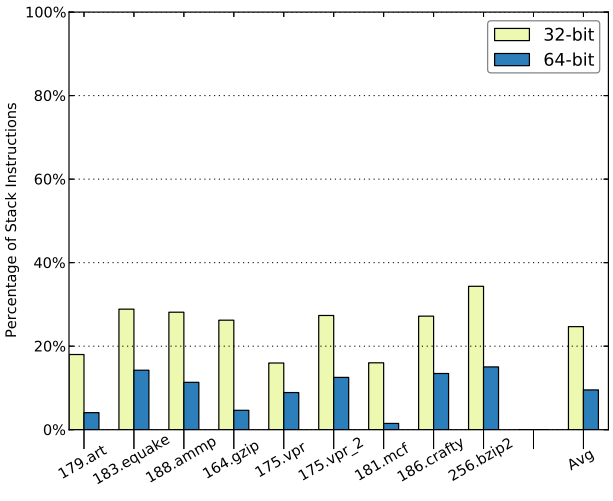


Fig. 1: The percentage of stack instructions of total dyanmic instructions for 32-bit and 64-bit x86 binaries.

II. Motivation

Specialized architectures are unburdened by the requirements of legacy ISAs, and a significant part of the efficiency gains from such architectures can be attributed to hardware specialization of the datapath, memory, and program control. ISA-independent analysis is attractive for such architectures because it avoids artificial constraints imposed by details of a specific ISA. Compilers for conventional ISAs must generate binaries that meet the specification of the instruction set semantics, and this process can alter the fundamental program behavior because of these constraints. This section discusses the three major sources of ISA constraints that we explore: overheads of stack operations due to register spilling, ISA-specific complex operators, and calling conventions.

A. Stack Overhead

Instruction set architectures support a finite number of registers which must necessarily be equal to or less than the number of physical registers in a machine. When writing code in a high-level language, most programmers are unaware of these constraints and use as many variables as the program requires. In order to fit the large number of variables into the ISA-defined register set, compilers must perform register allocation to map program variables to registers. When there are more variables that need to be allocated than available ISA-defined registers, the compiler will spill additional variables onto the stack, which is a specially reserved portion of the main memory. Load/store operations are inserted to manage the allocation of the machine registers and the stack. These stack memory operations can be expensive from a run-time performance point of view. For characterizing workloads for specialized architectures that do not have a fixed or known ISA, the stack accesses insert possibly unnecessary load/store operations into the instruction trace and incur additional memory utilization. These effects are not true program characteristics; they are artificial constraints imposed by the ISA.

We demonstrate the effect of stack operation by comparing 32-bit and 64-bit x86 binaries generated by LLVM's Clang compiler for a set of SPEC CPU benchmarks. One of the major differences between the 32- and 64-bit x86 ISAs is that 64-bit x86 has eight more general-purpose registers. Figure 1 plots the percentage of dynamic instructions that access the stack for 32-bit and 64-bit versions of SPEC benchmarks. We observe that for all of the benchmarks, the 32-bit binary has a much higher percentage of stack instructions than the 64-bit binary. This is because the additional general-purpose registers allow more variables to stay in registers, so less spilling to memory is required.

The stack overhead also applies to RISC ISAs. Lee *et al.* characterized stack access frequency using the Alpha ISA to propose a mechanism to separate stack from heap accesses[12]. For the same SPEC2000 workloads, they find a similar percentage of stack operation (24%) compared to our observation in 32-bit x86.

B. Complex Operations

We identify two classes of instructions as complex operations: vector instructions and compute or branch instructions with memory operands. Both kinds of operations can be split into multiple simpler primitives. CISC ISAs like x86 contain complex operations including vector instructions like SSE and instructions that support memory operands. We note that complex operations can exist even in RISC ISAs. For example, POWER and ARM include complex operations such as predicate instructions, string instructions, and vector extensions.

Most existing ISAs have already encoded some degree of specialization towards these complex operations by grouping multiple simple operations into single instructions. However, designers of specialized architectures may consider specialized functional units that combine sequences of operations into a single block. From a program analysis point of view, it is easier and cleaner to start from simple primitives and explore aggregation possibilities rather than to start from a more complex version of code resulting from another category of optimization.

We quantify the amount of complex operations in x86 in Figure 2. In this categorization, we treat an instruction as a complex operation if it is either a vector instruction (SSE) or a compute or branch instruction with a memory operand. The top three categories in Figure 2 are complex operations: vector operations, vector operations with memory accesses, and compute or branch instructions with memory operands. The remaining category includes all single operation instructions. We see that on average 27% of the total instructions executed are complex operations.

C. Calling Convention

The ISA calling convention describes how subroutines receive parameters from callers and how they return results. Any machine-dependent ISA needs to have its own specifications to pass arguments between subroutines. For example, x86,

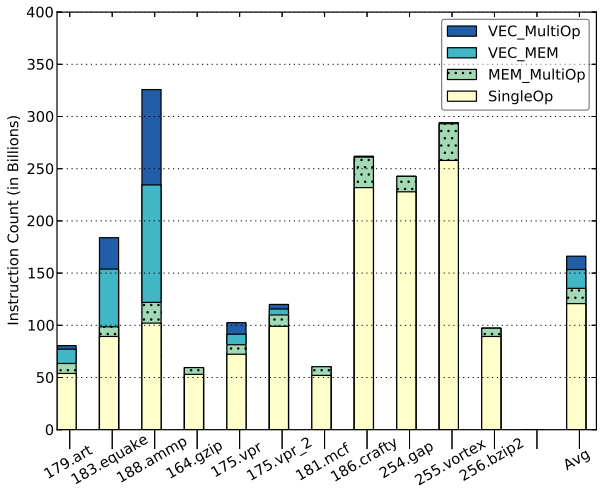


Fig. 2: Instruction breakdown of complex (top three bars) and single (bottom bar) operation instructions.

due to its limited number of registers, pushes arguments onto the stack before a subroutine is called, resulting in additional stack operations. Other ISAs also require various housekeeping operations for subroutines, and these are also artifacts of the ISA choice, not intrinsic to the behavior of the workload.

III. Methodology and Background

To evaluate the importance of performing workload characterization by using machine-independent code representation, we perform both ISA-independent and ISA-dependent analysis. After describing how these two analyses are performed, we provide details about how we sample benchmark executions and how we generate the code to be analyzed.

A. ISA-Independent Study

An ISA-independent representation of code is critical for the development of flexible compiler infrastructures, and modern compilers use ISA-independent intermediate representations to bridge high level source languages (e.g., C) to specific ISAs (e.g., Intel x86). Since our requirements for a code representation are similar to those of compilers, we leverage the intermediate representation used in compilers to perform our analysis.

Our analysis uses a specific intermediate representation (IR) available in the ILDJIT compiler [5]. Specifically, workload execution is represented by a trace of semantically equivalent ILDJIT IR instructions. This trace is generated by executing the IR code with a special-purpose interpreter that emits IR instructions as it executes them.

Compared to other possible intermediate representations available in mainstream compilers, like GCC and LLVM, the ILDJIT IR has the unique feature of being closer to source languages than to machine code. As we describe later in this section, this feature allows us to perform an analysis that is machine-, ISA-, and system-library-independent, so that workload-specific characteristics are exposed. Before describing the ILDJIT IR in detail, we motivate its use for our specific analysis by describing the unwanted consequences of relying on more standard intermediate representations used in mainstream compilers.

a) Compiler intermediate representation: Intermediate representations commonly used by compilers are either tree-based or linear. Tree-based representations like GCC's GIMPLE [2] are not designed to be executed to generate a run-time trace, making the implementation of an interpreter for such formats more challenging than for other representations. Implementing interpreters for linear representations, like LLVM's bitcode [3], is more straightforward, since the execution order to follow is given by the linear order imposed by the language itself.

Intermediate representations are a bridge from source languages to ISAs, but the ones used in mainstream compilers are closer to the latter. The rationale is that compilers are designed to maximize the number of code optimization algorithms that rely on the intermediate representation (both machine-independent and machine-dependent), and performing machine-specific optimization is easier if the representation is closer to the machine code.

The price paid for having a very low-level intermediate representation is that code analysis performed at this level (either at compile or run time) can be influenced by artifacts of either a specific ISA family or underlying system library implementations. For example, LLVM's bitcode language specifies the calling convention to use, and this code is included in the program representation. Moreover, these representations often do not identify source-language operations such as memory allocation, leading to platform-specific execution traces that can include artificial program behavior that is not intrinsic to the workload. For example, in ILDJIT, the allocation of an object is accomplished by a high level memory allocation operation. In contrast, compilers with low-level IRs often allocate objects using a generic call to the function *malloc* which is provided by the standard C library. In the latter case, a trace of the generated intermediate representation would include the code of the malloc function, and this can lead to artificial program dependencies between malloc invocations. On many common implementations of the malloc function, e.g. many Linux-based systems, the return address is computed based on a value of a local variable of that function. This local variable keeps the address of a free memory location in internal memory of the C library; just before returning the allocated address to the caller, this local variable is updated to point to another free memory location. Hence, this implementation creates a read-after-write chain of dependencies among different invocations of malloc. Notice that this dependence chain is not intrinsic to the considered workload; it depends on the specific implementation of the C library in use in the current system. On the other hand, by providing a memory allocation operation in the intermediate representation, platform-specific details about how memory is allocated are hidden, allowing analysis to be

system-independent.

b) ILDJIT IR: ILDJIT is a modular compilation framework that includes both static and dynamic compilers. As mentioned earlier, it includes a high-level intermediate representation (IR). ILDJIT performs a large set of classical, machine-independent optimizations at the IR level including copy propagation, dead-code elimination, loop-invariant code motion, etc. When the IR code is fully optimized, it is translated to LLVM's bitcode language and LLVM's back ends are used to optimize the code using machine-dependent optimizations and to generate semantically equivalent machine code.

We customized ILDJIT to implement an ad-hoc interpreter of its intermediate representation to emit IR instructions as they are executed. The IR instructions interpreted are the ones used for translation to the bitcode language. By attaching our interpreter right before the translation to bitcode, we ensure that the IR is fully optimized; however, machine-dependent information is still not used for these optimizations, allowing our analysis to study workload-specific characteristics.

The ILDJIT IR is a linear machine- and ISA-independent representation that includes common operations of high-level programming languages like memory allocation (e.g., new, free, newarray) and exception handling (e.g., throw, catch). It is a RISC-like language in which memory accesses are performed through loads and stores. Each instruction has a clear and simple meaning where only scalar variables, memory locations, and the program counter are affected by their execution. The language allows an unbounded number of typed variables (virtual registers), making analysis independent of the number of physical registers. Moreover, parameters of method invocations are always passed by using variables, as in the input source language we use (C), making analysis independent of specific calling conventions. Finally, the data types described in the source language are preserved in the IR language, making this representation closer to the input language compared to other compiler intermediate representations.

IR instructions that perform operations among variables require homogeneity among their types: an add operation between variables x and y requires the same type for both x and y (e.g., 32-bit integer). This characteristic leads to instructions that convert values between types. Notice that these conversions are required by the workload as the semantics of operations in the source language specify them. However, some of these conversions are unnecessary if a CISC-like ISA is used instead of the ILDJIT IR. Finally, opcodes (e.g., add, mul) are orthogonal with data types (e.g., integer, floating point). This opcode polymorphism constrains the number of different instructions in the language to 80, allowing an easy parsing of the executed trace.

B. ISA-Dependent Study

We perform our ISA-dependent analysis using the x86 instruction set. The x86 ISA is commonly used in architecture studies, and a large number of program analysis tools are available for workload characterization. For analysis of new x86-based microarchitectures, architects must understand the ISA-specific effects of the architecture since they can have a significant impact on pipeline and memory system design. When considering new heterogeneous architectures with both x86 and specialized cores, it would be natural to use existing workload characterization approaches. However, when performing workload characterization of specialized architectures, x86 provides a particularly poor starting point, because of the overheads discussed in Section II. In this study, we compare x86 instruction trace with ILDJIT IR trace. To generate the trace of x86 instructions executed by the workload, we use Pin, a dynamic binary instrumentation tool developed by Intel[13].

C. Sampling

Because of storage and processing time constraints, performing some of the analysis presented in this paper on the full execution trace is impractical. Therefore, we sample the execution with SimPoint[16]. We configure SimPoint to generate 10 phases, each of which contains 10 million instructions. Only instructions that belong to the identified phases are emitted and then analyzed.

In order to perform a fair comparison between x86 and IR traces, we sample the execution with the IR trace by configuring SimPoint to use IR instructions rather than the x86 ones. Then we instrument the code to identify the x86 instructions semantically equivalent to the IR code for the identified phases. In this way, we ensure that the same code region is considered for both the IR and x86 analysis.

D. Benchmark Suite

We use C benchmarks from SPEC CPU2000 benchmark suite. These benchmarks are translated to CIL bytecode by the compiler GCC4CLI [1] (a branch of GCC), and then they are compiled to IR by ILDJIT. Finally, ILDJIT generates the machine code by relying on LLVM's x86 back end as previously described. ILDJIT currently only supports the 32-bit LLVM back end and all of the results in the paper are for 32-bit operations.

IV. Workload Characteristics Analysis

In this section, we compare x86 and ISA-independent IR-based program analysis. We compare the two approaches using three main categories: **Compute**, **Control**, **Memory**. Table 1 summarizes the metrics that we compare in this section. The choice of metrics is intended to highlight opportunities for hardware specialization. Compute, control and memory are the most important metrics to represent workload characteristics and help designers gauge the complexity of specialization.

A. Compute

Specialized hardware often exploits custom functional units that combine multiple operations with predictable control flow in order to execute code more efficiently. Example of this approach is Conservation Cores [17], which identifies the hot functions in a program's execution and designs hardware accelerators for those functions. In order to uncover the opportunity to find sequences of operations that are amenable to

Category	Analysis	Notes
Compute	Opcode	Unique Opcodes required to cover 90% of dynamic instructions
Memory	Total Memory Footprint	Total number of unique memory addresses accessed
	90% Memory Footprint	Number of unique memory addresses that cover 90% of memory accesses
	Global Memory Address Entropy	Measure of the randomness of memory addresses
	Local Memory Address Entropy	Measure of the spatial locality of memory addresses
Control	Unique Branch Instructions	Total number of unique branch instructions
	Branch Entropy	Measure of the randomness of branch behavior, representing branch predictability

Table 1: Metrics used for workload characterization.

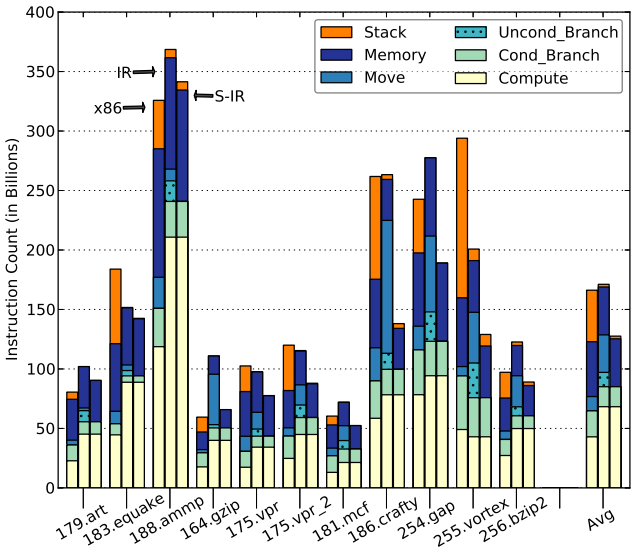


Fig. 3: The instruction breakdown for x86, IR and Simplified-IR (S-IR).

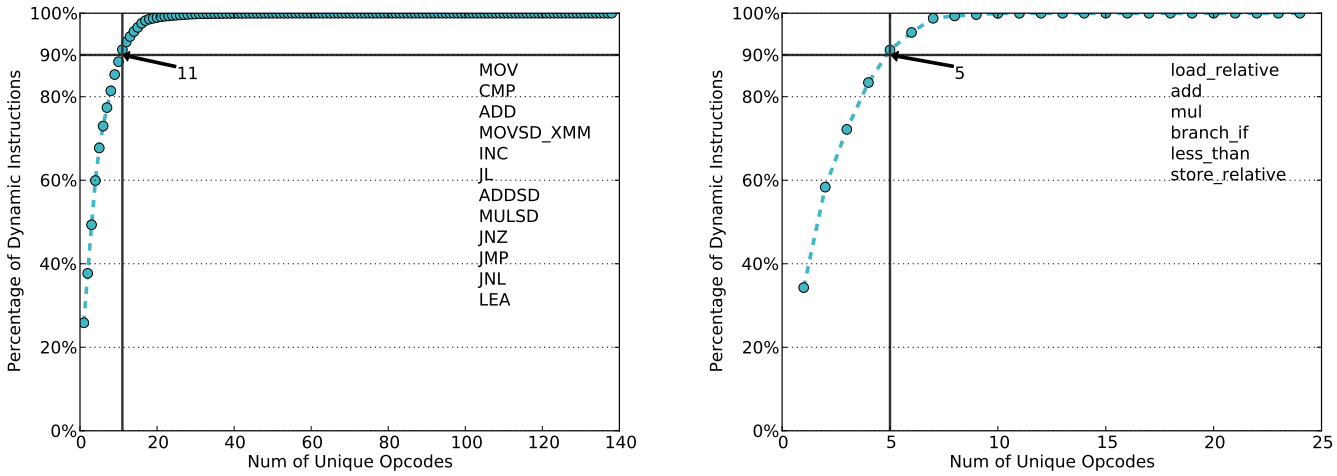
similar specialization, we need to analyze executed instruction sequences and detect various patterns. For such analysis, the way the operations are represented in the instruction trace will have a significant impact on whether certain patterns can be found or not and, subsequently, whether the workload is worth the effort of custom hardware design. In this section, we analyze the instruction breakdown and the most common opcodes found in both x86 and IR. We observe that x86 incurs more overhead for the basic computation performed by the application.

1) Instruction Breakdown: We start the analysis by categorizing the executed instructions from the IR and x86 code. We split instructions into the following categories: Stack, Memory, Move (data movement and conversion between registers), Unconditional Branch, Conditional Branch and Compute. Figure 3 shows this breakdown. For each benchmark, the left most bar represents the x86 binary, and the middle bar represents IR. Furthermore, during our implementation, we found that there is also instruction overhead associated with IR characteristics that are not intrinsic to the workloads. One source of such inefficiency is the number of unconditional branch instructions. The ILDJIT compiler does not remove these instructions because the compiler back end performs unconditional branch removal in a very efficient manner. Another source of overhead is data movement and conversion between registers. Such instructions appear in both IR and x86 and are used to support different data types and simplify optimizations. The right most bar in Figure 3 is what we call Simplified-IR – the IR trace without those two classes of instruction. In our following discussion, "IR trace" will refer to this simplified IR.

Consistent with the results from LLVM's 32-bit Clang compiler in Figure 1, we see that the number of stack-referencing instructions can be significant depending on the application. This is represented by the top section of the left most bar for every benchmark. For example, almost half of the x86 instructions for 255.vortex use the stack, while the effect is less obvious for benchmarks like 179.art. More importantly, the large number of stack accesses is caused by constraints of the x86 ISA (a small register set) and is not part of the intrinsic program behavior. This is evident from the IR bars for the stack-heavy benchmarks – moving from x86 to the infinite-register IR significantly decreases the number of accesses to the stack. While stack effects can increase the number of executed x86 instructions, CISC x86 instructions can combine multiple primitive operations together. This results in a more compact execution. For example, for benchmarks like 164.gzip and 179.art there are more instructions in the IR trace compared to the x86 one. The presence of x86-specific effects that both increase and decrease executed instructions makes it even harder to extract ISA-dependent overhead and expose the workload's intrinsic behaviors, further strengthening the case for analysis on the IR level.

2) Opcode Diversity: Our next experiment examines the diversity of the opcodes in the x86 and IR traces. Opcode diversity is relevant since it is related to the complexity of customized functional units in specialized hardware. Fewer and simpler opcodes will simplify the design of such hardware because the functional units will be more modular and reusable. This allows sharing such functional units across various workloads.

In order to compare x86 and IR analysis, we profile the total number of opcodes and the number of times each single opcode occurs in the program execution. We do not differentiate opcodes based on addressing modes, which reduces the number of required x86 opcodes. Figure 4 plots the number of unique opcodes and the percentage of dynamic instructions those opcodes cover for the benchmark 179.art. The dotted line on the plot shows the cumulative distribution of opcodes needed to cover the dynamic execution of the program.



(a) x86

(b) IR

Fig. 4: Cumulative distribution of the number of unique opcodes of 179.art. The intersecting lines show the number of unique opcodes that cover 90% of dynamic instructions.

To meaningfully compare x86 and IR, we use a horizontal line to highlight the number of unique opcodes required to cover 90% of the dynamic instrucitons in Figures 4a and 4b. This metric is meaningful for accelerator studies since it allows comparison of the number of functional unit types needed for different workloads. The horizontal line intersects with the cumulative distribution function to show the required number of opcodes. The x86 results demonstrate that 90% of the execution can be covered by 11 unique opcodes, while the same analysis with IR requires only 5 opcodes. The right portion of the plots shows the top opcodes used for both instruction sets. For x86, two MOV instructions, MOV and MOVSD_XMM, and four different conditional jump instructions are required. Compared with x86, the top opcodes from IR analysis are much clearer – the 5 opcodes are all simple primitives, resulting in a much simpler representation of the actions of the program.

We extend this comparison to all available benchmarks in the suite and show the result in Figure 5. Not surprisingly, for all the benchmarks the x86 trace needs more unique opcodes than the IR trace. Furthermore, the right most bar in Figure 5 shows the number of unique opcodes required to cover all benchmarks we analyze, computed as a superset of individual benchmark needs. In order to cover all the benchmarks in x86, 40 unique instruction opcodes are required; but the IR-based analysis uncovers only 12 fundamental primitives. Thus, extracting workload pieces that are amenable to hardware specialization appears significantly easier on the IR level of abstraction.

3) Static Instructions: The diversity of opcodes represents the different types of fundamental computing blocks that custom hardware might require. Another important metric is the number of static instructions required to cover the dynamic execution. In a custom design, different sequences of static instructions will lead to more or less complex data flow. Similar to the metric we use for opcode analysis, we compare the number of unique static instructions required to cover 90% of the dynamic instructions. As shown in Figure 6, benchmarks like 186.crafty and 255.vortex with significant stack overhead require more unique static instructions. The x86 characterization hides the truly important instructions, instead highlighting the stack overhead operations. This potentially misleads the identification of hot computation.

B. Memory

Memory behavior is crucial for workload performance. In the case of hardware specialization, the memory system must be tuned to the workload characteristics in order to realize significant gains in efficiency. In this section, we compare two memory characterization metrics, memory footprint size and memory entropy. We once again discover that ISA-dependent analysis can be significantly misleading and obscure the workloads' intrinsic behavior.

1) Memory Footprint: The first metric we consider is the size of the data memory that a program uses, including both stack and heap memory. We look into two types of memory footprint. The first one is the full memory footprint – the total size of data memory the program has accessed. It quantifies the overall memory usage. The second metric identifies the "important" memory footprint, which we define as the number of unique memory addresses that covers 90% of dynamic data memory accesses. This metric shows the most frequently used addresses that need to be kept close to the computation.

Figure 7 shows the total memory footprint analysis. The Y-axis in this figure is the number of unique memory addresses generated. The x86 and IR memory footprints are nearly the same, because total working set is intrinsic to the workloads and therefore independent of the program representation.

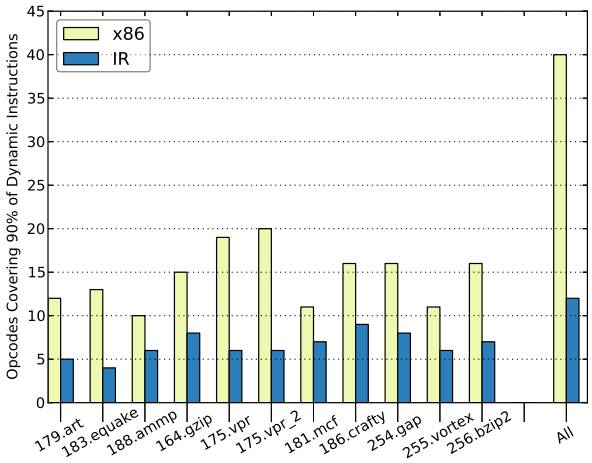


Fig. 5: Number of unique opcodes to cover 90% of dynamic instructions. "All" represents the global superset.

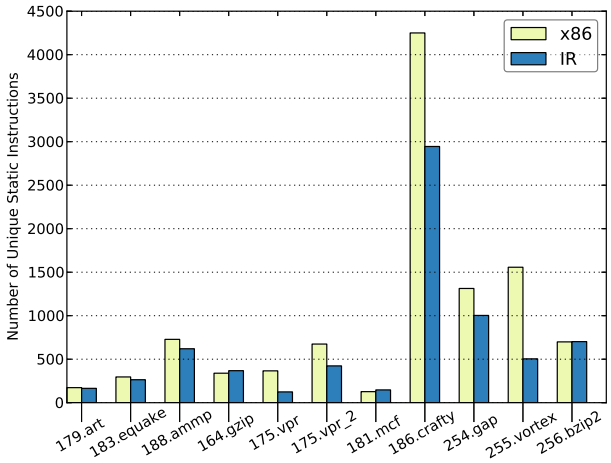


Fig. 6: Number of unique static instructions to cover 90% of dynamic instructions.

However, the important memory footprint, shown in Figure 8, has markedly different characteristics between x86 and IR. In most cases, there are fewer unique memory addresses needed for x86 compared to IR: although the total number of unique memory addresses is similar between x86 and IR, the memory accesses of x86 are dominated by a small number of addresses. The reason for this once again lies in frequent accesses to the stack. While the memory space of stack addresses is usually small, these addresses are accessed very frequently. When identifying important memory addresses, the few stack addresses that are frequently accessed will stand out and dominate the memory behavior. Thus, the important memory addresses found will be an artifact of the ISA instead of the program behavior.

2) Memory Address Entropy: We introduce the metric of **memory address entropy** in order to quantify how easy it is to keep memory data close to the computation. Intuitively, memory address entropy quantifies the information content, or the lack of predictability, in memory accesses. Thus, it is a metric opposite of memory locality that is often exploited by custom hardware – locality measures the amount of structure in memory addresses, while entropy measures its lack. We show that ISA-level analysis exposes a lower amount of entropy, leading to false assumptions of memory access structure.

a) Entropy: In information theory, entropy [15] is used to measure the randomness of a variable, which is calculated as Equation 1

$$Entropy = -\sum_{i=1}^{N} p(x_i) * log_2 p(x_i) \tag{1}$$

where $p(x_i)$ is the probability of x_i, N is the total number of samples of the random variable x. The result, $Entropy$, is a measure of predictability of the next outcome of x. For example, assume the pattern of variable x is very regular – always 1. In this case, $p(1) = 1$ and $log_2 p(1) = log_2 1 = 0$, so $Entropy = 0$, which means that it is very easy to predict x. One the other extreme, if there are N possible outcomes of x occurring equally often, $p(x_i) = \frac{1}{N}$. According to Equation 1

$$\begin{aligned} Entropy &= -\sum_{i=1}^{N} p(x_i) * log_2 p(x_i) \\ &= -N * \frac{1}{N} * log_2(\frac{1}{N}) \\ &= log_2 N \end{aligned} \tag{2}$$

which is very high for large N.

Yen, *et al.*, describes the idea of using entropy to represent the randomness of instruction addresses[18]. According to Equation 1, in the case of memory entropy, variable x represents the memory addresses that appear in the program execution. The probability $p(x_i)$ is the frequency of a specific memory address x_i. After profiling the unique memory addresses accessed in the workloads and the number of times each address is referenced, we can compute the memory address entropy of the workloads. When the memory entropy is high, the memory access stream is more random and less amenable to architecture techniques that require locality. Conversely, if the entropy is low, memory accesses are very regular and easier to predict.

b) Global Memory Address Entropy: Global memory entropy describes the randomness of the entire data address stream using all address bits (32 in our case). Figure 9 shows the calculated global memory address entropy for both x86 and IR. For each benchmark, the leftmost bar is the global entropy of x86 memory addresses. The rightmost bar is the case for IR memory addresses. We can see that the entropy of x86 is generally much lower than for IR. In order to find the reason for the difference, we compute the x86 memory address entropy without the stack addresses, shown in the middle bar. As we can see, after removing the stack addresses, the x86 address entropy is comparable with the IR memory address entropy, which represents the intrinsic address randomness of the workloads.

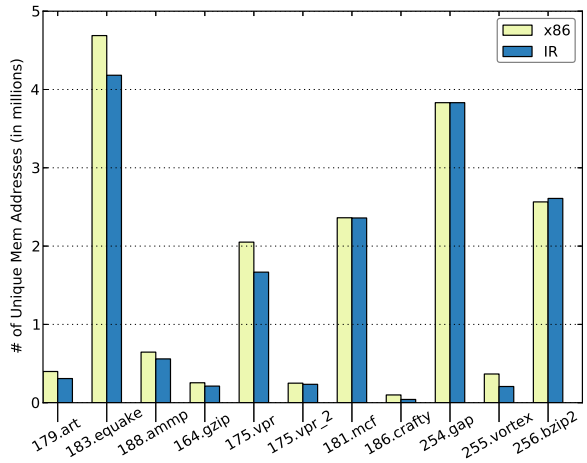


Fig. 7: Number of unique memory addresses to cover 100% of dynamic memory accesses.

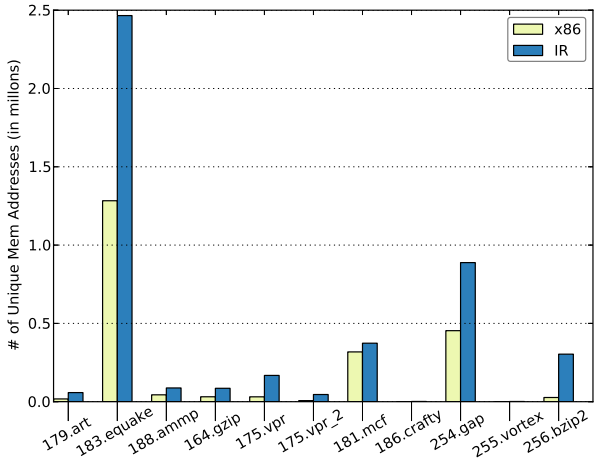


Fig. 8: Number of unique memory addresses to cover 90% of dynamic memory accesses.

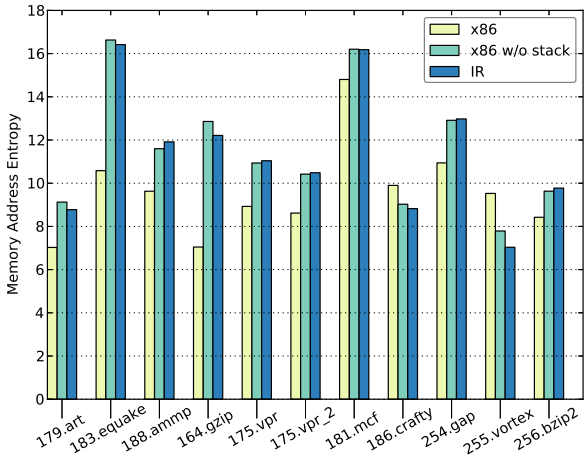


Fig. 9: Memory address entropy of x86, x86 w/o stack, IR traces. Lower values indicate more regularity in the access stream.

From this we can see, the ISA overhead has a big impact on the memory address behavior.

c) Local Memory Address Entropy: Local memory entropy computes the address entropy using a subset of the entire address bits. Local entropy can help us detect spatial locality in the workloads. For example, we can skip the lower-order bits of the addresses, and compute entropy only with the high-order address bits, as seen in Figure 10a. If the local address entropy with, for example, 28 bits shrinks significantly compared to global entropy, memory accesses are less random, and significant spatial locality is present. After we ignore the lower order bits, such spatial locality is expressed by grouping those addresses that are close together.

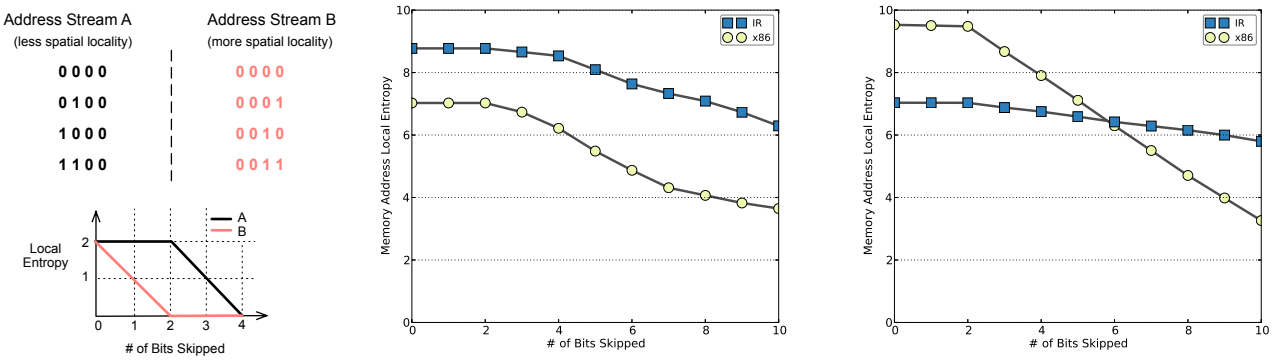
Figure 10 shows two examples of the local address entropy when we sweep the number of low order bits ignored from 0 to 10. The two benchmarks, 179.art and 255.vortex, are representative of the patterns we have seen among the rest of the benchmark suite. For both cases, the local entropy of x86 drops faster than for IR. This is very obvious for 255.vortex. This is due to the fact that stack addresses are usually in close proximity, which means they have usually have good locality. Ignoring the lower-order bits results in steeper drops in entropy for x86. This also shows ISA-dependent analysis will bias the workload characteristics towards better locality due to the impact of stack operations.

C. Control

Control flow complexity is a very important metric for workload characterization. From our experience in general purpose processor design, we know that speculative execution is necessary to exploit parallelism. In a heterogeneous architecture, there maybe a variety of cores or computing engines with different degrees of support for speculation. In order to choose the appropriate ones to run the workloads, the control complexity of the workloads needs to be fully understood and not dependent on a specific architecture. In this section, we compare the control complexity analysis of x86 and IR and show that both analyses are consistent with each other, showing that ISA choice has a minimal effect on a workload's control flow.

1) Branch Instruction Count: Our first-order control flow analysis counts the number of unique conditional branch instructions to cover 90% of the branches. This is similar to the unique opcode analysis but focused on branch instructions. This is important for hardware specialization because it measures the number of control flow decisions that must be handled in a design.

As Figure 11 shows, the number of unique branch instructions to cover 90% dynamic branches is consistent between



(a) Example of Local Entropy (b) 179.art (c) 255.vortex

Fig. 10: Local memory entropy as a function of low-order bits omitted in calculation. A faster dropping curve indicates more spatial locality in the address stream.

x86 and IR. Both sets of bars track each other very well. This implies that ISA choice does not have a significant impact on the number of branch instructions generated, which mostly depends on the way programs are written.

2) Branch History Entropy: Another important metric is control flow predictability, which is intrinsic to the workload. Generally speaking, if the branch taken patterns are more regular and less random, branches are easier to predict. In this sense, the degree of the regularity of the branch behavior will indicate the predictability of the control flow. Based on this intuition, Yokota proposed the idea of Branch History Entropy using Shannon's information entropy idea to represent a program's predictability[19].

We use a string of bits to encode taken or not taken branch outcomes. In this sense, the program as the producer of the sequence can be viewed as an information source and we can compute the entropy of the information source to represent the regularity of branch behavior. In our implementation, we use a sequence of n consecutive branch results as the random variable and compute the entropy of the benchmarks. The results are shown in Figure 12. We can see that the branch entropy from x86 and IR also track each other very well. This shows that both ISA-dependent analysis and ISA-independent analysis fully expose the program's control behavior. This matches our intuition that ISA does not affect control flow significantly.

D. Workload Characterization Using ISA-Independent Characteristics

We compare the eleven SPEC benchmarks with five ISA-independent metrics from our analysis: the number of opcodes, the value of branch entropy, the value of memory entropy, the unique number of static instructions (I-MEM), and the unique number of data addresses (D-MEM). In terms of specialized architecture design, smaller values for each of these metrics indicate more regularity in the benchmarks and better opportunity to exploit specialization. For each metric, we choose the maximum value across all the benchmarks and for each benchmark we plot the relative value with respect to this maximum value. We generate kiviat plots for all benchmarks, shown in Figure 13, in which each axis represents one of the ISA-independent characteristics. The plot in the lower, right corner of the figure provides a legend for the individual axis. The kiviat plots are ordered by the area of the resulting polygon. With an equal weighting of the five characteristics, area provides a rough approximation for overall benchmark regularity (smaller area is more regular). We observe very different behavior across the benchmark suite. For example, 255.vortex demonstrates regularity across all the metrics, while 186.crafty has relatively low regularity in most of the dimensions. These insights will be helpful for specialized architecture designers to identify the opportunity for acceleration.

V. Related Work

A. Microarchitecture-Dependent Characterization

There has been a significant amount of prior work on using performance counters to understand and optimize the performance of workloads [8], [14], [4], [7]. These studies use highly microarchitecture-dependent metrics like cycles per instruction, cache miss rate, and branch misprediction rate. These studies are helpful in finding performance bottlenecks on various platforms for different benchmarks. However, the characteristics profiled are biased by the microarchitecture the workloads are running and the target ISA.

B. Microarchitecture-Independent Characterization

Hoste and Eeckhout propose metrics of characterizing benchmarks based on microarchitecture-independent characteristics [11]. They instrument program binaries to profile characteristics like instruction mix, ILP, working set size, and branch predictability. They demonstrate that the program characteristics from performance counter style characterization can be misleading. As an example of the utility of these metrics, Eeckhout, *et al.*, demonstrate that microarchitecture-independent characteristics can be used to determine benchmark similarity, with the goal of sampling representative programs from the benchmarks suite [6]. In both cases, Alpha ISA traces are used to analyze microarchitecture-independent behavior.

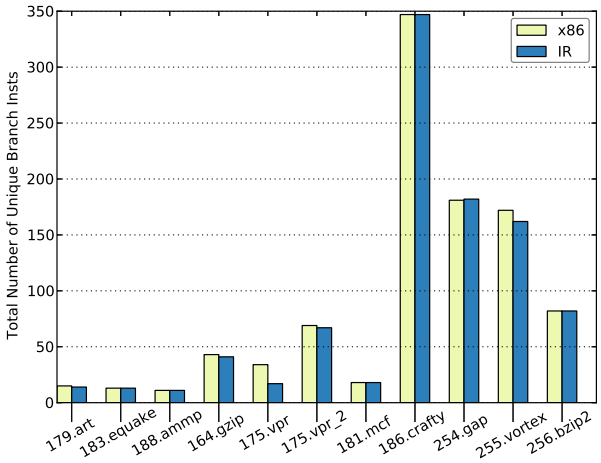


Fig. 11: Number of unique branch instructions to cover 90% of dynamic branches.

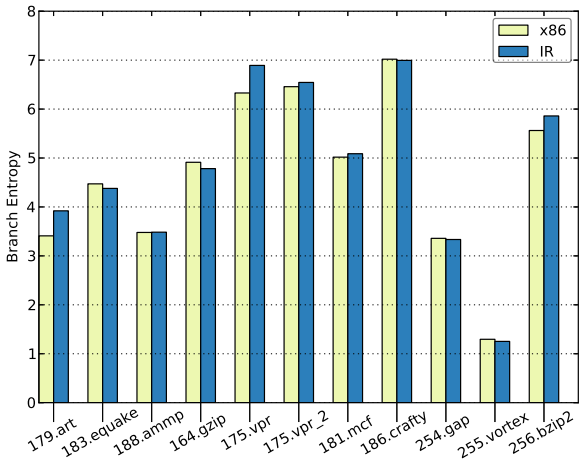


Fig. 12: Branch entropy per workload. Lower values imply better branch predictability.

C. Specialized Architecture

Specialized architectures offer large potential for performance and energy improvements. Hameed *et al.* explore the sources of performance and energy overheads in general-purpose processors by quantifying the overheads of a H.264 encoder running on a general-purpose system[10]. They identify that fully specialized hardware can be 500x more energy efficient than general-purpose processors, and in order to achieve ASIC-like energy efficiency designers need to apply customized storage and functional units tuned to the specific application. Triggered by the potential efficiency benefits, several research efforts have investigated the approach of specialized architecture design. For example, Venkatesh *et al.* proposed accelerating irregular hot functions using specialized C-Cores[17]. Another example is DySER[9], which integrates specialized functional units into a general-purpose processor's pipeline. Unlike other studies, our characterization tool is not intended to build specialized cores, but to provide an ISA-independent workload characterization that designers can use to more easily design accelerators. This is analogous to the role that machine-dependent workload characterization plays in helping designers develop microarchitectural extensions for general-purpose machines.

VI. Conclusion

This paper presents a new workload characterization approach targeting specialized architectures. Existing characterization approaches do not differentiate intrinsic workload characteristics from microarchitecture- and ISA-influenced program behavior. Specialized architectures can be radically different from traditional designs, including fixed function accelerators without ISAs. These designs have very different compute and memory behavior, and conventional workload characterization includes artifacts that can mislead designers of specialized architectures. In our study, we use the ISA-independent property of a compiler IR to profile ISA-independent characteristics of workloads. We compare both ISA-dependent and ISA-independent approaches for several analyses that are likely to be highly relevant for developing specialized architectures, including computation, memory behavior, and control flow. We discover that ISA-dependent characterization is misleading in identifying the intrinsic characteristics of the workloads. In particular, we discover that stack overhead due to the ISA can significantly bias the memory behavior which is crucial for workload characterization. We also perform workload characterization using the ISA-independent characteristics and show that this characterization can be helpful to guide accelerator designers towards opportunity for hardware specialization. Overall, ISA-independent optimization can identify the intrinsic characteristics of workloads and discover opportunities for hardware acceleration.

Acknowledgments

We thank Simone Campanoni, Glenn Holloway, Svilen Kanev, Gu-Yeon Wei and the anonymous reviewers for their useful feedback and insights related to this work. This work was supported in part by C-FAR, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This work was also partially supported by the National Science Foundation (NSF) Expeditions in Computing Award #: CCF-0926148. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

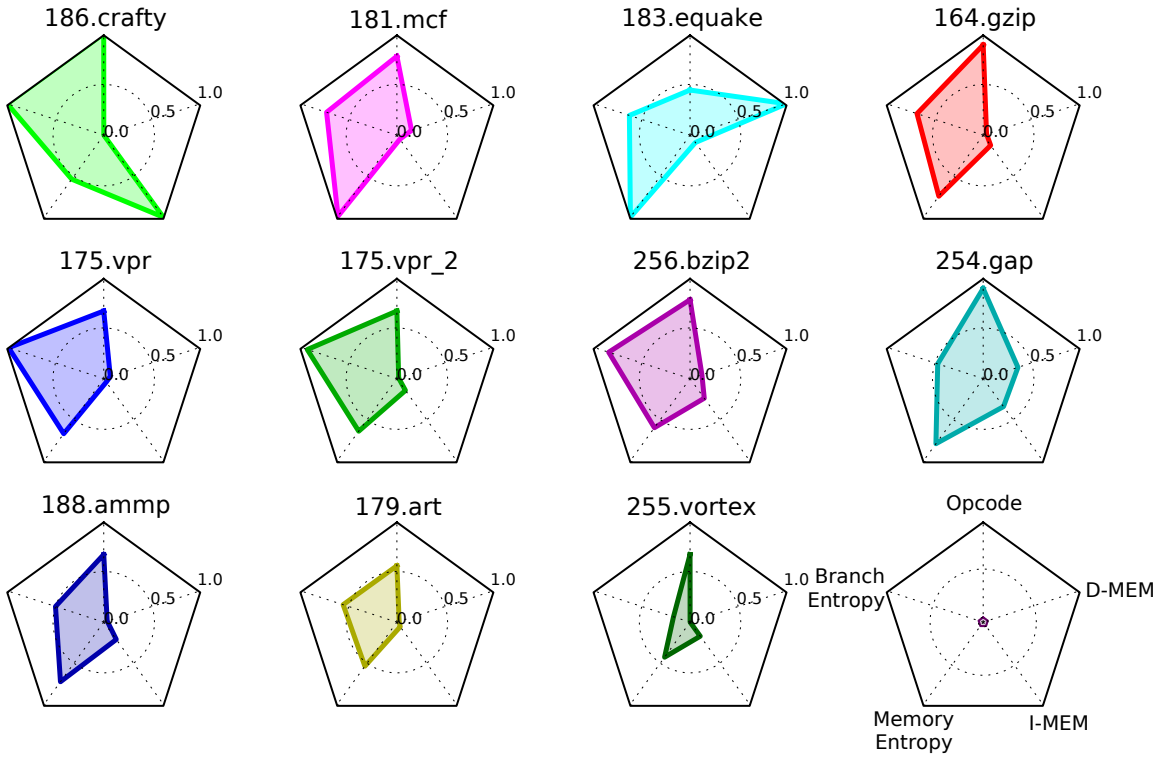


Fig. 13: Comparison of five ISA-independent metrics across SPEC benchmarks, ordered by the area of the polygon. The lower right kiviat plot provides the legend, and smaller values indicate more regularity in the metric.

REFERENCES

[1]GCC4CLI. http://gcc.gnu.org/projects/cli.html.

[2]GNU compiler collection (GCC) internals, GIMPLE documentation. http://gcc.gnu.org/onlinedocs/gcc-4.3.0/gccint/index.html.

[3]LLVM assembly language reference manual, bitcode documentation. http://llvm.org/docs/LangRef.html.

[4]S. Bird, A. Phansalkar, L. K. John, A. Mercas, and R. Idukuru. Performance characterization of SPEC CPU benchmarks on Intel's Core microarchitecture based processor. In *SPEC Benchmark Workshop*, 2007.

[5]S. Campanoni, G. Agosta, S. Crespi-Reghizzi, and A. D. Biagio. A highly flexible, parallel virtual machine: Design and experience of ILDJIT. *Softw. Pract. Exper.*, 2010.

[6]L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *International Symposium on Workload Characterization*, 2005.

[7]L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 2003.

[8]K. Ganesan, L. John, V. Salapura, and J. Sexton. A performance counter based workload characterization on Blue Gene/P. In *International Conference on Parallel Processing*, 2008.

[9]V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *International Symposium on High Performance Computer Architecture*, 2011.

[10]R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *International Symposium on Computer Architecture*, 2010.

[11]K. Hoste and L. Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. In *International Symposium on Workload Characterization*, 2006.

[12]H.-H. S. Lee, M. Smelyanskiy, C. J. Newburn, and G. S. Tyson. Stack value file: custom microarchitecture for the stack. In *International Symposium on High Performance Computer Architecture*, 2001.

[13]C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2005.

[14]T. K. Prakash and L. Peng. Performance characterization of SPEC CPU2006 benchmarks on Intel Core 2 Duo processor. In *International Conference on Parallel Processing*, 2008.

[15]C. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 1948.

[16]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*, 2002.

[17]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*, 2010.

[18]L. Yen, S. C. Draper, and M. D. Hill. Notary: Hardware techniques to enhance signatures. In *International Symposium on Microarchitecture*, 2008.

[19]T. Yokota, K. Ootsu, and T. Baba. Introducing entropies for representing program behavior and branch predictor performance. In *Workshop on Experimental Computer Science*, 2007.