Instruction Set Usage Analysis for Application-Specific Systems Design

Charles Mutigwe
School of Electrical and Computer Systems Engineering
Central University of Technology
Bloemfontein, South Africa 9320
Email: cmutigwe@ieee.org

Johnson Kinyua
School of Computer Information Systems
Virginia International University
Fairfax, VA 22030 USA
Email: jkinyua@viu.edu

Farhad Aghdasi
Faculty of Science and Agriculture
University of Fort Hare
Alice, South Africa 5700
Email: faghdasi@ufh.ac.za

***Abstract*—The manner in which the resources of a microprocessor are used affects its performance, power consumption and size. In this work we show how increasing the size of a processor's instruction set, in turn, increases the amount of hardware needed to implement that processor. We also study how efficiently the hardware resources of four processor architectures are used by measuring the static instruction set utilization of a group of benchmark applications. The architectures examined are the Intel x86, Intel x86-64, MIPS64, and PowerPC. We introduce the notions of instruction subsets, exact cores and general-purpose cores, and then we use these concepts to propose a new measure of processor resource utilization, core density. Based on the core density measure we show that on average 9 exact cores are equivalent to a single general-purpose core in the existing architectures and that in particular instances this multiplier can go up to 48 exact cores.**

***Index Terms*—Computer architecture, ISA, system-on-a-chip, core density, design methodology**

I. Introduction

A number of empirical studies on instruction set usage [1]–[4], including our study presented in this paper, have shown that most of the instructions in the instruction sets of microprocessors are rarely used by the applications using those microprocessors. If a microprocessor's instruction set is a measure of the hardware resources needed to implement that microprocessor, then these results from the empirical studies suggest that most of the hardware resources on microprocessors with a fixed instruction set architecture (ISA) are being used inefficiently.

The instruction set of a processor serves as an interface between the processor's hardware and the software applications set to run on that hardware. To the software programmer, the instruction set exposes a processor's functionality, while to the processor designer it is a measure of the hardware resources that will need to be implemented in the processor. Given two processors with the same bus size, but with different architectures; the one with a larger instruction set will expose more features or operations to the software applications and it will often require more hardware resources (as will be shown later in this paper) and design effort to implement.

The rest of the paper is organized as follows. Section II discusses some related research efforts. In Section III we discuss the relationship between a processor's instruction set and the hardware resources needed to support that instruction set. We also define the core density measure. Section IV describes the instruction set usage experiments, the results of the experiments and discusses their implications. Section V concludes the paper.

II. Related Work

Foster *et al.* [1] describe two types of instruction set usage analyses. In the first type, the static case, the frequency counts of the instructions used to specify the logic of the problem are collected and analyzed [1], [2]. In the second type, the dynamic case, the frequency counts of the instructions used to execute the logic of the problem are collected and analyzed [3], [4]. Foster *et al.* [1] also proposed two measures for instruction usage. The first measure is based on information theory and it calculates the average number of bits of information contained in each opcode for each application that is analyzed. The second measure estimates the effort needed to recode an application when the number of opcodes available to the compiler or assembly programmer is reduced after an initial unconstrained compilation. Furthermore, Foster *et al.* [1] found that the hand-assembled code has higher static opcode usage than machine-compiled code, and that there are no significant differences in dynamic opcode usage between the hand-assembled and machine-compiled code.

Empirical studies by Ibrahim *et al.* [2], Adams and Zimmerman [3], and Huang and Peng [4] on the x86 instruction set architecture (ISA), and those by Hennessy and Patterson [5] on several CISC and RISC ISAs, have shown that modern applications spend 80-90% of their time accessing only 10-20% of the ISA. The studies on the x86 ISA also found that static usage of the top 25 opcodes accounted for more than 90% of the total number of opcodes in the applications. In this paper, we will use the terms *instruction* and *opcode* interchangeably.

III. Instruction Sets and Hardware

Let R be the set of all the hardware resources of the processor. These resources are represented by functional units. Let I be the processor's instruction set, where:

$$R = \{r_1, \ldots, r_M\} \quad \text{and} \quad I = \{i_1, \ldots, r_N\} \tag{1}$$

Let P be the power set of R. For any i where $i \in I$, let R_i represent the set of resources needed to implement the instruction i and

$$R_i \in P \quad \text{and} \quad R_i \subset R \tag{2}$$

Let R_I represent the resources needed to implement all the instructions in I, then

$$R_I = \bigcup_{i=1}^{N} R_i \tag{3}$$

Resources that are not directly related to the implementation of any instruction, such as the pipelining microarchitecture, will be treated as a constant and referred to as M_{const}, where:

$$M_{const} = |R - R_I| \tag{4}$$

As the number of shared resources decreases, that is as

$$\left| \bigcap_{i=1}^{N} R_i \right| \to 0,$$

the processor's performance improves. An example of this relationship is the improved performance of directly-implemented processors versus their microprogrammed versions.

For directly-implemented processors there is a one-to-one function f, such that $f : I \mapsto P$ and as the number of instructions in I increases, so does $|R_I|$.

In the case of microprogrammed processors f is not one-to-one, several instructions may map onto one set of resources. However, each instruction represents a distinct operation and this distinction is captured in the microinstructions used to describe it. All the sets of microinstructions and the lookup tables matching them to their corresponding instructions are stored in the control memory. Winiewska *et al.* found that as the number of microinstructions was increased by a factor of 3, the amount of control memory required to store them increased by a multiple of 54 [6], suggesting an exponential growth in hardware requirements. Jian-Lun notes that for some processors (including the Intel x86), the control memory takes up 50% of the area on the chip [7].

We conclude that regardless of whether a processor is directly-implemented or it is microprogrammed: **as the number of supported instructions ($|I|$) increases, then the hardware resources needed to implement the processor ($|R_I|$) increases**.

$$|I| \propto |R_I| \tag{5}$$

A. Instruction Subsets

Each application that executes on a processor uses a set of instructions, which we will refer to as the application instruction set, A. The application instruction set is a subset of the processor instruction set. The resources, R_A, required to implement the application-specific processor for A are,

$$R_A = \bigcup_{i=1}^{|A|} R_i, \text{ where } R_A \subset R_I \tag{6}$$

From the subset relationship $A \leqslant I$ and using (6), we have

$$|R_A| \leqslant |R_I| \tag{7}$$

The results of the experiments presented below show that for the benchmark applications, $A \ll I$, so using this along with (5) and (7) we have:

$$|R_A| \ll |R_I| \tag{8}$$

That is, the average application-specific processor requires much less hardware resources than the processor that implements the complete instruction set. The next section describes a measure that can be used to compare the resources between the two processors in (8).

We will refer to the processor that implements the complete processor instruction set as a *general-purpose core* (**GPC**). While, the processor that implements only the application instruction sets that will target it, we will call an *exact processor core* (**EPC**). There is another type of core that allows for run-time extensions to the GPC, which we refer to as the *extensible processor core* (**XPC**). The Xtensa processor from Tensilica provides an example of an XPC [8]. In every case, the $GPC \subset XPC$, the ISA utilization of the XPC is at best only equal to that of the GPC, so in this study we will only address the relationship between the GPC and the EPC.

B. Core Density

Given a set of applications that have been compiled to run on a given ISA, we define the applications' core density (η), as:

$$\eta = \frac{\text{hardware to implement a single GPC}}{\text{hardware to implement an EPC for application set}} \tag{9}$$

Rewriting (9), we have:

$$\eta = \frac{\alpha|I| + M_{const}}{\alpha|A| + M'_{const}} \tag{10}$$

TABLE I
ARCHITECTURES STUDIED

Architecture	Unique Opcodes
MIPS64	1182
PowerPC	533
x86	659
x86-64	1101

where α is the ratio of the number of bits required to encode I to those required to encode A. The core density measure, through α, is related to the two measures of instruction set usage proposed by Foster *et al.* [1].

In order to estimate M_{const} and M'_{const}, we assume that 50% of the processor consists of instruction-dependent hardware. This estimate is in line with Jian-Lun's findings [7]. Using this approximation (10) reduces to:

$$\eta = \frac{|I|}{|A|} \tag{11}$$

IV. Experimental Results

In our experiments we only considered opcodes and their static usage. We varied four factors; the benchmark applications, the ISA of the target processors, the compilers, and the compiler optimizations. Below is a description of how the experiments were setup.

A. Platforms & Benchmark Applications

A virtual machine with the 64-bit version of the Linux Ubuntu 10.40 LTS distribution was used as the platform for all the experiments. We used the GNU Compiler Collection (*gcc*) and the Portable C Compiler (*pcc*) to build the C benchmark applications. *gcc* and *pcc* cross-compilers were built for the hardware processors shown in Table I.

The benchmark applications consist of the ten C benchmark applications that are part of the SPEC CPU2006 benchmark suite [9].

All the benchmark applications were compiled three times with the *gcc* compilers and with each compilation a different optimization scheme was used together with the *-S* option to generate assembler files instead of binary files. In the first iteration there is no optimization, in the second the applications were optimized for size (*-Os* flag), and in the final iteration they were optimized for speed (*-O3* flag). All the applications were, again compiled twice with the *pcc* compilers and with each compilation a different optimization option was used. In the first *pcc* iteration there is no optimization, in the second the applications were optimized for speed (*-O* flag).

B. Results and Discussions

1) Compiler: The same source code for the benchmark applications was used with the *gcc* and *pcc* compilers. These two compilers follow different compilation strategies, as displayed by the relative differences in the instruction types of the compiled code in Fig. 1.

However, with regards to the relationship between instruction set utilization and architecture, the choice of compiler

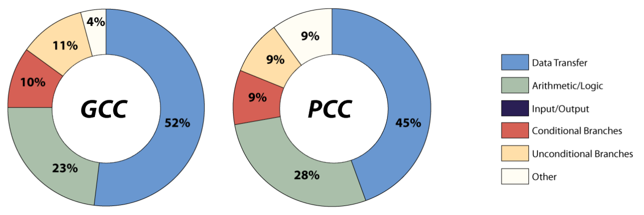

Fig. 1. Instruction type usage by compiler.

did not make a significant difference as can be seen from Fig. 2. The average utilization for all the applications in the C benchmark set is on average 2% higher than when the applications are considered individually. The PowerPC and x86 architectures have higher utilization rates compared to the MIPS and x86-64 architectures. The latter architectures have larger instruction sets.

2) Optimizations: While the compiler optimizations affected the program size, we observed that they did not have a significant impact on the ISA utilization rate, as shown in Fig. 3. With the exception of the PowerPC architecture, the differences between any pair of results in the ISA utilization-versus-compiler optimizations all fall within a 3% margin. The margin for the PowerPC architecture is 6%. Based on this observation, from this point forward we will only discuss the results for the applications that were optimized for size.

3) Instruction Set Usage: The ISA utilization results for the hardware processors, expressed in terms of core density, are presented in Table II. Considering the case of the x86-64 processor, we note that if it were used to execute any one of the ten applications its average core density is 21.5. However, if it is only executing the 999.specrand benchmark application its core density is 47.9, indicating that for this application the traditional processor model or GPC leaves more than 98% of the processor resources idle, since only 1 exact core is required, but resources equivalent to 48 exact cores are deployed in the existing traditional processor.

4) TopN Instructions: The top 25 instructions account for more than 89% of all the instructions used by the benchmark

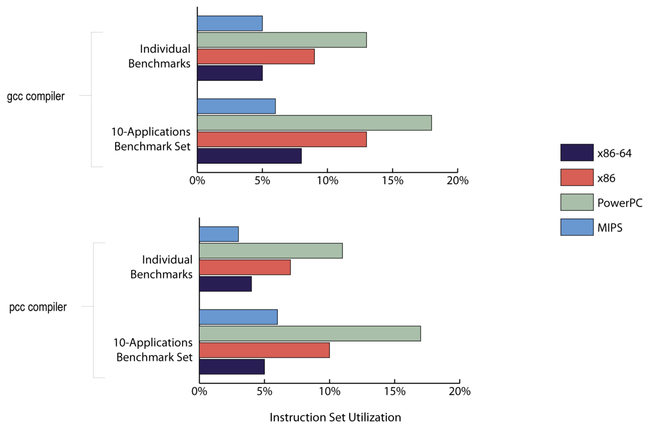

Fig. 2. Instruction set utilization by compiler and architecture.

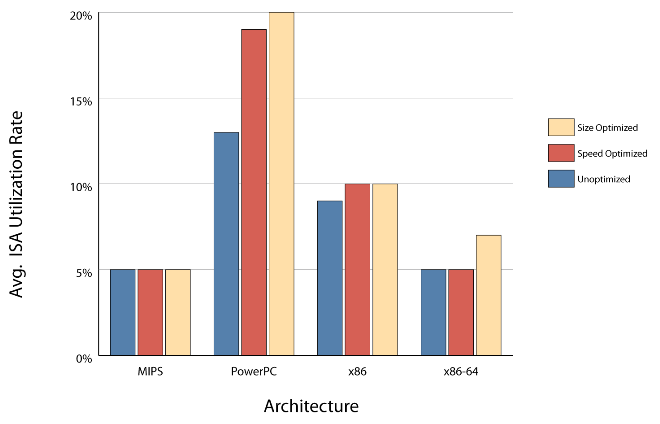

Fig. 3. Instruction set utilization by optimization.

applications, Figure 4. This result confirms the locality of reference property of applications and is in line with the results of Adams and Zimmerman [3], and Hennessy and Patterson [5]. The top 100 instructions account for all the instructions used by all the processors. This leads us to question *why most current processors have more than 100 instructions*, which is a reframing of the CISC-RISC debate. A follow-up question is *which 100 instructions are needed?*

Our answer is to keep any existing ISAs, but to provide two options to create application-specific processors. In the first option, the instruction set usage of the applications that are scheduled to run on the processor is performed and the list of unique opcodes used is extracted. The applications may be pre-existing application binaries, which are disassembled to get the unique opcodes or they are applications compiled from source code. An application-specific processor is then synthesized using the list of used opcodes; this processor can be referred to as a subset processor. In the second option, a processor that supports a pre-determined subset of the ISA is implemented without knowing beforehand the applications that are going to use the subset processor. Next, a constrained compiler targeting this subset processor is developed. Finally, the applications set to run on the subset processor are (re)compiled.

V. Conclusion

Our study results indicate that the average resource utilization of modern fixed ISA microprocessors is in the 5-20% range. We show that on average nine heterogeneous exact cores can be placed on the same microprocessor chip where currently a single general-purpose core resides. As part of this study we define a new measure, core density, to describe the relationship between application-specific exact cores and general-purpose cores. Static resource utilization is application-dependent and the results show that by using exact processor cores the resulting system can be as much as 48 times more efficient than today's processors.

TABLE II
Application Core Densities

Application\Architecture	MIPS	PowerPC	x86	x86-64
401.bzip2	20.38	5.03	10.14	19.32
403.gcc	16.42	3.37	7.40	13.59
429.mcf	30.31	8.20	16.07	29.76
433.milc	20.74	5.38	8.45	18.35
456.hmmer	16.19	4.26	7.57	13.43
458.sjeng	17.91	5.03	9.98	18.35
464.h264ref	16.19	3.73	7.01	13.59
470.lbm	23.64	8.46	12.43	25.60
482.sphinx3	16.65	4.33	8.04	14.68
999.specrand	39.40	15.68	24.41	47.87
Average	**21.78**	**6.35**	**11.15**	**21.45**
η/10-App Set	**15.76**	**3.05**	**6.72**	**11.35**
Maximum η/10-App Set	**39.40**	**15.68**	**24.41**	**47.87**

References

[1] C. C. Foster, R. H. Gonter, and E. M. Riseman, "Measures of op-code utilization," *IEEE Transactions on Computers*, vol. 20, no. 5, pp. 582–584, 1971.

[2] A. H. Ibrahim, M. B. Abdelhalim, H. Hussein, and A. Fahmy, "Analysis of x86 instruction set usage for Windows 7 applications," in *2nd International Conference on Computer Technology and Development (ICCTD)*, 2010, pp. 511–516.

[3] T. L. Adams and R. E. Zimmerman, "An analysis of 8086 instruction set usage in MS DOS programs," *SIGARCH Computer Architecture News*, vol. 17, no. 2, pp. 152–160, 1989.

[4] I. J. Huang and T. C. Peng, "Analysis of x86 instruction set usage for DOS/Windows applications and its implication on superscalar design," in *Proceedings of the International Conference on Computer Design (ICCD)*, 1998, pp. 566–573.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA: Morgan Kaufmann, 2002.

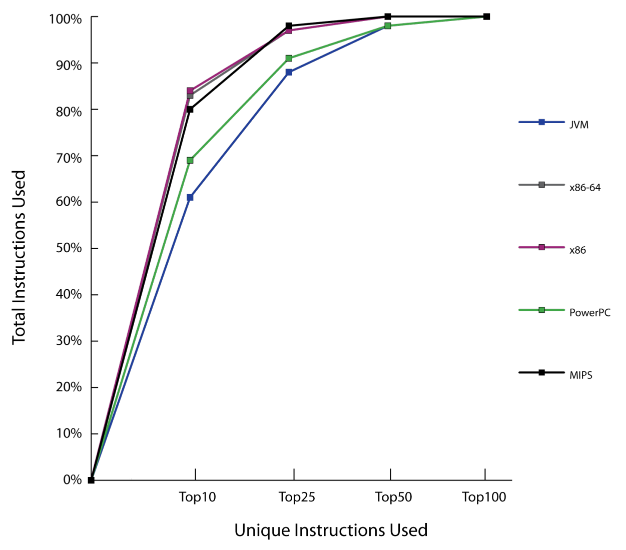

Fig. 4. Cumulative instruction usage by architecture.

[6] M. Wisniewska, M. Adamski, R. Wisniewski, and W. A. Halang, "Application of hypergraphs in microcode length reduction of microprogrammed controllers," in *Proceedings of the 2nd International Workshop on Nonlinear Dynamics and Synchronization (INDS)*, 2009, pp. 106–109.

[7] S. Jian-Lun, "Researches on the technology of high performance microprogrammed control," in *Proceedings of the International Conference on Educational and Information Technology (ICEIT)*, 2010, pp. V2:20–V2:24.

[8] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[9] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.