Characterizing Embedded Applications for Instruction-Set Extensible Processors

Pan Yu
panyu@comp.nus.edu.sg

Tulika Mitra
tulika@comp.nus.edu.sg

School of Computing
National University of Singapore
Singapore 117543

ABSTRACT

Extensible processors, which allow customization for an application domain by extending the core instruction set architecture, are becoming increasingly popular for embedded systems. However, existing techniques restrict the set of possible candidates for custom instructions by imposing a variety of constraints. As a result, the true extent of performance improvement achievable by extensible processors for embedded applications remains unknown. Moreover, it is unclear how the interplay among these restrictions impacts the performance potential. Our careful examination of this issue shows that significant speedup can only be obtained by relaxing some of the constraints to a reasonable extent. In particular, to the best of our knowledge, ours is the first work that studies the impact of relaxing control flow constraint by identifying instructions across basic blocks and indicates 5–148% relative speedup for different applications.

Categories and Subject Descriptors: C.1.3 [Other Architecture Styles] Adaptable architectures

General Terms: Algorithms, Performance, Design.

Keywords: Customization processors, Instruction-set extensions.

1. INTRODUCTION

The application-specific nature of embedded system creates new opportunities to customize processor architecture for a particular application. This customization is critical in meeting the challenging demand on performance without incurring high cost in terms of energy and area. A recent trend in customization is to extend an existing processor core with a set of **custom instructions**. These custom instructions in the extended Instruction Set Architecture (ISA) can be implemented in the processor's datapath itself or as a separate co-processor. Tensilica's Xtensa [7], Altera's NIOS [1], and Lx [6] are some examples of commercial extensible processors. Given the growing interest in the area, this paper examines the limits of performance gain due to extended ISA for embedded applications.

A custom instruction is simply a fragment of the program's dataflow graph mapped onto a hardware **Custom Functional Unit (CFU)**. It encapsulates the computation of a cluster of primitive instructions and helps achieve several benefits including improved code size, register pressure, and execution cycles. However, certain characteristics of the core processor, cost considerations, and compiler limitations restrict the size, shape and number of custom instructions.

- **Number of Operands:** The base architecture of the core processor may impose constraints on the maximum number of source and destination operands used by the custom instructions. The length of a custom instruction increases with increasing number of operands which may be difficult to accommodate in the standard format of the base ISA. Moreover, the number of input and output ports to the register file is proportional to the number of input and output operands required by an instruction. The cost and energy consumption of a processor increase significantly with increasing number of register file ports. These considerations may impose limits on the maximum number of operands.

- **Number of custom instructions:** The format of the base ISA may limit the number of custom instructions that can be introduced. For example, if the base ISA implements 26 instructions using fixed length 5-bit opcode, then it can accommodate six new instructions.

- **Area:** As cost is a major consideration for embedded systems, only a limited amount to die area is expected to be available for the implementation of the CFUs.

- **Control Flow:** Custom instruction identification is typically performed within basic block boundaries. (Basic block is a code fragment with single entry and exit points.) The assumption is that the compiler cannot exploit instructions that cross basic block boundaries.

We investigate the performance impact of these constraints on extensible processors for embedded applications. We observe that in real life it is impossible **not** to put any such constraint. Some of them are imposed by the designers of the base processor core. Others are imposed artificially by the tools that automate the ISA extension process in order to make the problem more tractable. To the best of our knowledge there has not been any systematic study of the effect of these restrictions. Therefore, it is not clear how much theoretical performance potential exists for extensible processors and how the different constraints are interacting with each other to limit that potential. We do so in this paper in the same spirit as the paper by Wall [16] that examines the limits of instruction level parallelism for general-purpose programs. The goal of our work is to offer some guidelines to the designers on the relative importance of the different constraints. The designers can then avoid putting in a constraint which is too restrictive to allow the extensions to achieve the performance potential.

2. RELATED WORK

Commercial examples of extensible processors include HP Laboratory and STMicroelectronics' Lx [6], Altera's NIOS [1] and Tensilica's Xtensa [7], whereas examples from academia are Chimaera [17] and AEPIC [15].

Automatic ISA extension generation consists of: (1) *Custom Instruction Identification* that identifies patterns[1] meeting certain topology requirements and (2) *Custom Instruction Selection* that selects the most important patterns under resource and other constraints. Almost all the custom instruction identification methods attempt to identify patterns within a basic block. The only exception is [2] which identifies patterns based on execution trace and can possibly go beyond basic blocks even though their actual implementation does not. Pozzi et. al. [13] imposes further constraints on the instruction topology by allowing only Multiple Inputs Single Output (MISO) patterns and thereby achieving linear complexity in the number of instructions. Similarly, Goodwin [8] imposes a limit of 2-input 1-output, single-cycle execution in order to identify all possible patterns within a basic block. The problem in identifying Multiple Input Multiple Output (MIMO) patterns is that there can potentially be exponential number of them corresponding to a basic block. Arnold et. al. [2] avoids this problem by using an iterative technique that first identifies 2-operator patterns, replace their occurrences in the DDG, and repeats the process. However, this technique might get stuck in a local maxima. Moreover, as the approach is trace-based, its time/space complexity is very high. Atasu et. al. [3] searches a full binary tree and decides at each step whether or not to include a particular instruction in a pattern. The potential exponential search space is pruned based on input/output constraints. Recently, Clark et. al. [5] has proposed a heuristic method to speedup the exploration process. Given the set of candidate patterns, [3] proposed an optimal method to select at most N patterns. Both ILP-based [11] and heuristic-based [5] methods have been proposed for pattern selection under area constraints. Finally [2] presents a dynamic programming approach to maximize the performance gain when there is no constraint.

The work in [2, 3, 5, 6, 11, 13] all show potential performance gain due to custom instructions. However, all these works make some assumption or the other in both the identification as well as the selection process. This is not unexpected given the inherent complexity of the problem. However, the assumptions make it unclear how the choice of different constraints impacts the performance improvement due to ISA extensions. This papers takes a holistic approach towards this problem and explores the limit of performance potential for extensible processors. In that respect, our work is in the same spirit as that of Wall [16], which examines the amount of instruction-level parallelism existing in a program at the limits of feasibility and even beyond.

[1]In this paper, we will use the terms custom instruction and patterns interchangeably

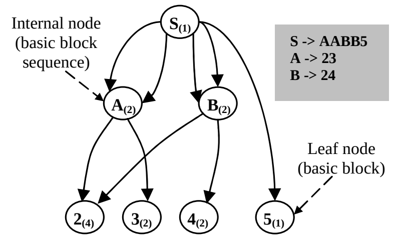

Figure 1: WPP for basic block sequence 232324245 with execution count annotations

3. METHODOLOGY

As mentioned in Section 2, ISA extension identification consists of custom instruction identification and custom instruction selection. In this section, we describe our methodology and explain how we overcome the exponential blowup that arises especially when we do not impose any constraint.

3.1 Custom Instruction Identification

Most of the research in custom instruction identification are based on analyzing the basic blocks (program fragment with single entry and exit points) in isolation. The only exception to this is [2] which identifies custom instructions based on the dynamic execution trace of the program. As we are interested in identifying the performance potential of customization at the limits of feasibility, our identification process is also based on dynamic execution trace. This way we can also identify custom instructions and their frequencies across basic block boundaries. However [2] constitutes a huge data dependence graph for the entire trace and builds patterns incrementally by traversing this graph multiple times. This approach is computationally expensive thereby limiting it to small patterns. Instead our study is based on a compact representation of the dynamic execution trace called Whole Program Path (WPP) [10] that allows us to identify patterns within and across basic blocks in an efficient manner.

Whole Program Path (WPP). Larus developed the notion of Whole Program Path (WPP) [10] which captures the entire execution trace of a program. The storage overhead for the trace is reduced drastically by employing online string compression techniques called SEQUITUR [12]. SEQUITUR algorithm [12] represents a finite string σ (the control flow trace in our case) as a context free grammar whose language is the singleton set $\{\sigma\}$. The execution path of a program can be viewed as a string (over an alphabet of basic blocks) from which the grammar is synthesized *on-the-fly*. The time complexity of the algorithm is linear in the length of the input string. The grammar is represented as a directed acyclic graph, called WPP. Figure 1 shows an example of WPP. Each node of the WPP is annotated by the execution count of the sub-DAG rooted at that node.

Traversing WPP. The leaf nodes of the WPP are basic blocks and their execution counts represent the execution counts of the basic blocks. An internal node represents a path which is a sequence of basic blocks appearing in the execution trace. We first start with the basic blocks and identify patterns within the basic blocks. To identify patterns across basic block boundaries, we look at frequently occurring internal nodes and treat the sequence of basic block corresponding to that node as the unit for pattern identification process.

Pattern Identification. Given a basic block or a sequence of basic blocks, for which we want to identify the patterns, we first create a data dependence graph (DDG). Given this DDG, we identify all the possible subgraphs that meet the pre-defined criteria such as number of input/output operands. This identification process is based on [3] that searches a full binary tree and decides at each step whether or not to include a particular instruction in a subgraph. The potential exponential search space is pruned based on input/output constraints. We modify this search so as to identify only connected subgraphs of the the DDG. Each identified subgraph is associated with a latency and area requirement.

3.2 Custom Instruction Selection

Given the set of candidate subgraphs, we first identify the identical subgraphs using a modified version of the algorithm presented in [14]. All the identical subgraphs map to a single CFU or custom instruction; that is, a custom instruction has multiple instances. The execution frequencies of custom instruction instances are different and results in different performance gains. The selection process attempts to cover each original instruction in the code with zero/one custom instruction to maximize performance. We first provide an optimal solution based on ILP for instruction selection followed by heuristic methods.

3.2.1 Optimal Solution using ILP

Let us first define the variables. We have N custom instructions defined by $C_1 \ldots C_n$. A custom instruction C_i can have n_i different instances occurring in the program denoted by $c_{i.1} \ldots c_{i.n_i}$. Each instance has execution frequency given by $f_{i.j}$. Let R_i be the area requirement of the custom instruction C_i and P_i be the performance gain obtained by implementing C_i in hardware as opposed to software (given in number of clock cycles). Finally, we define binary variables $s_{i.j}$ which is equal to 1 if custom instruction instance $c_{i.j}$ is selected and 0 otherwise. The objective function maximizes the total performance gain using custom instructions:

$$max: \sum_{i=1}^{N}\sum_{j=1}^{n_i}(s_{i.j} \times P_i \times f_{i.j})$$

We optimize the objective function under the constraint that a static instruction can be covered by at most one custom instruction instance. If custom instruction instances $c_{i_1.j_1} \ldots c_{i_k.j_k}$ can cover a particular static instruction, then

$$s_{i_1.j_1} + \ldots + s_{i_k.j_k} \leq 1$$

In order to model the area constraint or the constraint on the total number of custom instructions, we first define the variable S_i. S_i is the binary variable which is equal to 1 if

```
Input: Set of all custom instruction instances $X$; Area
       constraint $R$
Output: Instructions selected; Performance gain $P$
$P := 0$ ;
$S_i := 0 \;\; \forall 1 \leq i \leq N$;
$s_{i.j} := 0 \;\; \forall 1 \leq i \leq N, 1 \leq j \leq n_i$;
while $X \neq \emptyset$ do
    select the highest priority instr. instance $c_{i.j} \in X$;
    if $S_i = 1$ then
        $s_{i.j} := 1$;  $P := P + P_i \times f_{i,j}$;
    else if $R_i \leq R$ then
        $S_i := 1$;  $s_{i.j} := 1$;
        $R := R - R_i$;  $P := P + P_i \times f_{i.j}$;
        $K$:= static instructions belonging to $c_{i.j}$;
        remove all instances in $X$ containing $k \; \forall k \in K$;
    remove $c_{i.j}$ from $X$;
end
```

Algorithm 1: **Heuristic instruction selection method**

C_i is selected and 0 otherwise. S_i is defined in terms of $s_{i.j}$.

$$\begin{aligned} S_i = & \;\; 1 \quad \text{if} \sum_{j=1}^{n_i} s_{i.j} > 0 \\ = & \;\; 0 \quad \text{otherwise} \end{aligned}$$

However, the above equation is not a linear one. We substitute it with the following equivalent linear equations.

$$\sum_{j=1}^{n_i} s_{i.j} - U \times S_i \leq 0$$

$$\sum_{j=1}^{n_i} s_{i.j} + 1 - S_i > 0$$

where U is a large constant greater than $max(n_i)$.

If R is the total area budget for all the CFUs, then

$$\sum_{i=1}^{N}(S_i \times R_i) \leq R$$

Similarly, if M is the constraint on the total number of custom instructions, then

$$\sum_{i=1}^{N} S_i \leq M$$

3.2.2 Heuristic Methods

As the ILP based custom instruction selection may become computationally expensive for large number of custom instruction instances, we design heuristic algorithms as well. The idea is to assign priorities to the custom instruction instances. The instances are chosen starting with the highest prioritized one. We use the following three priority functions: (1) Performance/Cost ratio: $Priority(c_{i.j}) = (P_i \times f_{i.j})/R_i$, (2) Software execution time: $Priority(c_{i.j}) = (software_i \times f_{i.j})$ where $software_i$ is the total execution cycles if $c_{i.j}$ is implemented in software, and (3) Speedup: $Priority(c_{i.j}) = (P_i \times f_{i.j})$. The first priority function is more suitable under tight area budget whereas the third one maximizes performance gain when area is not an issue. The second one simply attempts to speedup the time consuming

Benchmark	**Class**	**Total BB**	**Hot BB**	**Avg. Hot BB Size**
rawcaudio	Telecomm	68	22	2.6
rawdaudio	Telecomm	66	18	2.6
fft	Telecomm	129	24	6.8
sha	Security	76	6	17.2
strsearch	Office	148	4	6
qsort	Automotive	30	26	4.9
bitcnts	Automotive	79	13	12.4
basicmath	Automotive	94	28	6
patricia	Network	203	37	2.8
dijkstra	Network	77	6	5
djpeg	Consumer	317	96	6.8

Table 1: Characteristics of benchmark programs

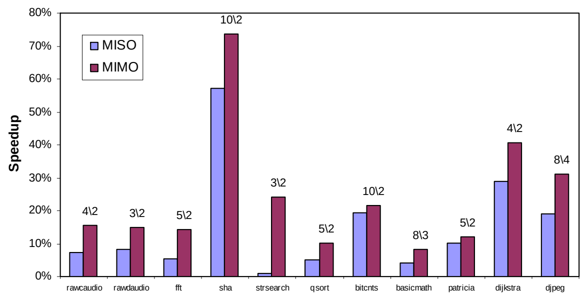

Figure 2: Comparison of MISO and MIMO.

portions of the applications. For each design point, we apply all the three heuristics and choose the one that gives the best result. Algorithm 1 describes the heuristic for a given priority function.

4. RESULTS AND ANALYSIS

In this section, we describe the findings of our limit study. Due to space constraints, only selected results are presented.

4.1 Experimental Setup

Table 1 shows the benchmark programs used in our limit study. All these benchmarks are from MiBench [9]: a free, commercially representative embedded benchmark suite. We have selected benchmark programs from all the different categories such as automotive, network, telecomm etc. We use the sample inputs for the benchmarks. Table 1 also shows the total number of basic blocks and *hot* basic blocks for each program. We define *hot* basic blocks as the ones whose aggregate contribution exceeds 95% of the total execution time of the program. Our ISA extension selection methodology only explores these hot basic blocks. Experimental results show that including patterns from the rest of the basic blocks does not improve performance. The average size of hot basic blocks varies from small (2.6 instructions) to quite big (17.2 instructions).

We generate the execution traces using Simplescalar tool set [4] which is a cycle-accurate simulation platform for RISC-like processor architectures. The benchmarks are compiled by gcc version 2.7.2.3 with -O3 optimization. We build the Whole Program Path (WPP) from the execution traces using a modified version of the Sequitur grammar [12]. We construct data dependence graphs (DDG) for the hot basic blocks and paths (internal nodes of WPP) to identify the custom instructions. We only impose the restriction that a pattern cannot contain memory operations. We consider integer-intensive benchmarks as including floating-point operators in patterns seldom results in speedup. For instruction selection, we use the ILP-based method when there is a restriction on the number of custom instructions and heuristic method under area constraint. The heuristic method chooses the optimal one most of the time under area constraint, but performs poorly under the total custom instruction constraint.

We calculate hardware latency/area for each of the simple operations in the Simplescalar ISA using Synopsys design tool with a popular cell library, based on which we approximate the latency/area of custom instructions. To calculate speedup, we use a single-issue, in-order pipelined architecture with 100% cache hit rate (benchmark inputs are the same as the profiling input). As most of the recent embedded processors, such as ARM11 and PowerPc602, are in-order processors, this is not an unrealistic assumption. The **Speedup** is given as

$$Speedup = (\frac{Exec.\ cycles\ with\ origninal\ ISA}{Exec.\ cycles\ with\ extended\ ISA} - 1) * 100$$

We first look at the speedup obtained by limiting the patterns to basic blocks. Later, we revisit the issues with patterns that can cross basic block boundaries.

4.2 Operand Constraint

The restriction on number of operands either comes from inherent limitations of the ISA or the register file design decisions of the base processor (e.g., Tensilica's Xtensa processor [7]). However, sometimes this is an artificial restriction imposed by the tool that automatically selects the extensions in order to prune the deign space [13, 8]. The most popular choices are (1) 2-input, 1-output patterns and (2) multiple-input single-output (MISO) patterns. In this section, we investigate how these choices affect the speedup due to extended ISA.

First, we restrict the patterns to 2-input, 1-output without imposing any other constraint. The results indicate that for most benchmarks, it is extremely difficult to find any such pattern. Even for benchmarks for which such patterns exist, the speedup is insignificant (maximum is around 3.2% for *dijkstra*). However, we observe that as we do not allow memory operations within a pattern, we cannot exploit 2-input, 1-output structures like $x = a[i]$.

Figure 2 shows the speedup for MISO and MIMO instructions without any other restrictions. Note that the speedup for MIMO represents the theoretical speedup obtainable when the patterns are restricted to basic blocks. The speedup is roughly 20% for most of the benchmarks except for *Sha* which achieves close to 74% speedup. There is an average of 9.1% speedup improvement by relaxing the single-output constraint imposed by MISO. The data labels for each MIMO bar show the minimum number of input and output operands required to obtain the maximum speedup. We observe that most benchmarks achieve the theoretical limit with only 2 output operands (for *djpeg* the speedup is only 0.8% less with 2 outputs).

As the number of output operands can be easily restricted to two according to Figure 2, we vary the number of input operands while the number of output operands is set to two (see Figure 3). Again there is no other restriction. With the exception of a few benchmarks, 4-input operands seem

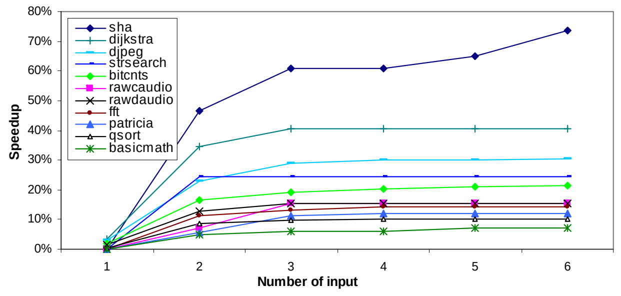

Figure 3: Effect of the number of input operands.

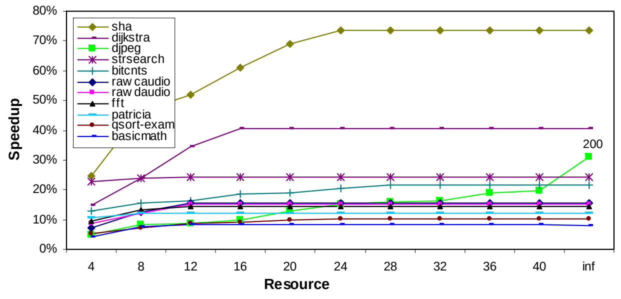

Figure 4: Effect of area constraint.

sufficient to achieve reasonable speedup. We conclude that even though 2-input, 1-output is quite a restrictive option, 4-input, 2-output can achieve close to the theoretical limit.

4.3 Area constraint

Given the cost conscious nature of embedded systems, it is likely that a chip-area budget will be imposed for implementation of the custom functional units (CFU) [5, 11]. Figure 4 shows the speedup with varying area budget and no restriction on number of input/output operands. The x-axis shows the resource budget in terms of number of adders. For most benchmarks, the resource requirement is very small — the area required to implement the custom instructions is roughly equal to that of 25 adders. The only exception is *djpeg* which requires area equivalent to around 200 adders for optimal speedup. In general, resource does not seem to be an issue for embedded benchmarks.

4.4 Total instruction constraint

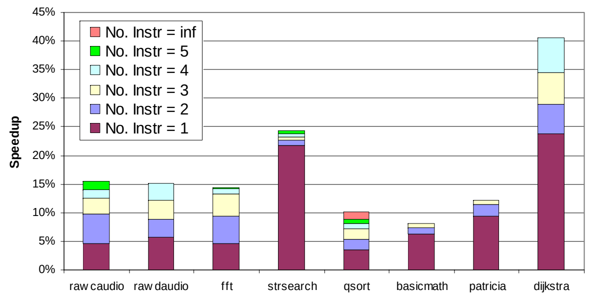

Figure 5: Effect of constraint on total number of custom instructions.

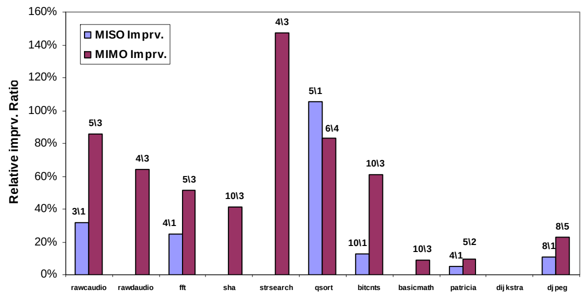

Figure 6: Relative performance improvement by relaxing control flow constraints.

Some extensible processors impose a limit on the total number of custom instructions that can be added. The tool developed by Atasu et. al. [3] finds the extended ISA that will achieve maximum speedup under this constraint. As shown in Figure 5, except for *qsort*, all the other benchmarks achieve maximum speedup under 5 instruction constraint. For the other three benchmarks (*sha*, *bitcnts* and *djpeg*), whose solutions cannot be obtained via ILP, heuristic methods indicate that the peak speedup requires 6, 10, 38 custom instructions respectively. With 5 instructions, they can achieve up to 78%, 74%, and 33% of peak speedup respectively. Five instruction constraint may not be good for some programs like *djpeg* whose datapaths vary a lot, but effective enough for most others to exploit the majority of the benefit from custom instructions.

4.5 Control flow constraint

A common restriction imposed by almost all the tools is that the patterns should be limited to basic blocks. The rationale being that it is hard for the compiler to exploit patterns that span multiple basic blocks. We study the performance potential that can be achieved by relaxing this constraint. Note that as we are using WPP to find hot paths (consisting of multiple basic blocks), we do not impose any artificial limit on the number of basic blocks in a path. However, our experiments indicated that for all the benchmarks opportunities exist only among 2–3 consecutive basic blocks. That is, it is quite local and attempting to find patterns across several basic blocks is not fruitful.

Figure 6 shows the effect of relaxing the control flow constraints for both MISO and MIMO (no area constraint). In general, compared to MISO, MIMO gets more improvement. The benchmark *dijkstra* does not get any improvement by allowing patterns to cross basic block boundaries. However, for others the relative improvement (relative to speedup of MISO and MIMO within basic blocks) ranges from a modest 5% to as much as 148%.

One question that may naturally arise is whether the resource consumption increases significantly as we cross basic block boundaries. Figure 7 shows the results for two selected benchmarks. In general, under tight resource budget, it does not help much to find patterns spanning basic blocks. For some values of the area constraint, speedup degrades across basic blocks; but that is an artifact of the heuristic. The total area budget requirement remains roughly the same irrespective of whether the patterns are within or across basic blocks. For the similar question about the effect of num-

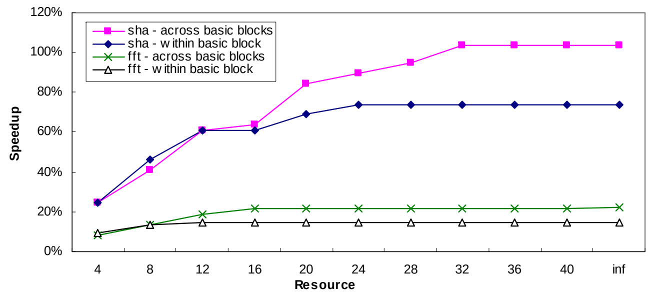

Figure 7: Speedup across basic blocks under varying area budgets for *Sha* and *FFT*

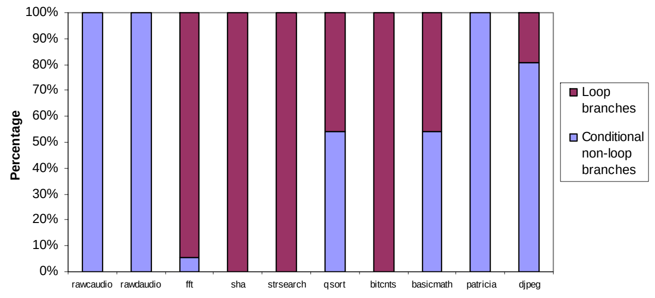

Figure 9: Contributions due to loop and conditional non-loop branches

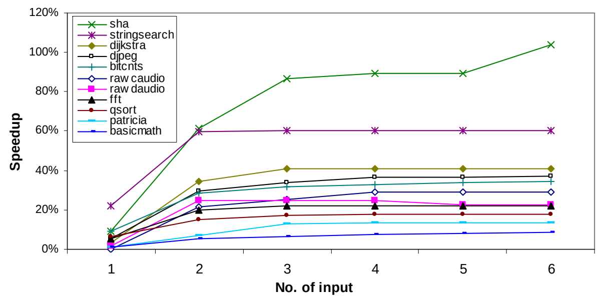

Figure 8: Effect of the number of input operands across basic blocks

ber of operands, Figure 6 shows that most benchmarks can achieve peak performance with 3 outputs. Under this restriction of 3 outputs, we show in Figure 8 that usually 4 to 5 inputs will suffice to obtain near optimal performance.

Finally, we discuss how compilers can exploit these patterns. A pattern spans basic blocks with either a loop branch or a conditional non-loop branch in between. The first case can be exploited through loop unrolling. For the second case, the compiler can combine the corresponding instructions from the basic blocks in question and add fix-up code for the situation where the branch is taken in the other direction. It can also use predicated execution if available. We investigate how much these two cases contribute to the performance gain in Figure 9.

5. CONCLUSIONS

We have studied the performance limit of extensible processors for embedded applications. Using a novel methodology based on compressed execution trace, we have calculated speedup for extended ISA under extremely relaxed conditions. The summary of our major findings are:

1. Relaxing control flow constraints can achieve 5–148% relative improvement without major impact on total resource requirement. Moreover, most of this improvement can be realized with existing techniques such as predication and loop unrolling.

2. One can put a reasonable limit on resource and number of custom instructions without affecting speedup.

3. Restrictions on number of operands (such as allowing only MISO or 2-input, 1-output patterns) can significantly limit the performance. However, 4-input, 3-output patterns achieve close to maximal speedup.

In future, we plan to extend this work to study the impact of ISA extensions on code size, register pressure, and energy.

6. ACKNOWLEDGMENTS

This work was partially supported by NUS research grants R252-000-088-112, R252-000-171-112 and A*STAR Project 022/106/0043.

7. REFERENCES

[1] Altera. Nios embedded processor system development. `http://www.altera.com/products/ip/processors/nios/nio-index.html`.

[2] M. Arnold and H. Corporaal. Designing domain-specific processors. In *CODES*, 2001.

[3] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *DAC*, 2003.

[4] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Toolset. Technical Report CS-TR96-1308, Univ. of Wisconsin - Madison, 1996.

[5] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *MICRO*, 2003.

[6] P. Faraboschi et al. Lx: a technology platform for customizable VLIW embedded processing. In *ISCA*, 2000.

[7] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2), 2000.

[8] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *CASES*, 2003.

[9] M. R. Guthausch et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001. `http://www.eecs.umich.edu/mibench/`.

[10] J. R. Larus. Whole program paths. In *PLDI*, 1999.

[11] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set design of configurable ASIPs. In *ICCAD*, 2002.

[12] C. Nevill-Manning and I. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7, 1997.

[13] L. Pozzi, M. Vuletic, and P. Ienne. Automatic topology-based identification of instruction-set extensions for embedded processor. Technical Report 01/377, Swiss Federal Institute of Technology Lausanne (EPFL), 2001.

[14] R.Kastner et al. Instruction generation for hybrid reconfigurable systems. *ACM Transaction on Design Automation of Electronic Systems*, 7(2), 2002.

[15] S. Talla. *Adaptive Explicitly Parallel Instruction Computing.* PhD thesis, New York University, 2000.

[16] D. W. Wall. Limits of instruction-level parallelism. In *ASPLOS*, 1991.

[17] A. Ye et al. Chimera: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ISCA*, 2000.