Limits of Data-Level Parallelism

Sreepathi Pai, R. Govindarajan, M. J. Thazhuthaveetil
Supercomputer Education and Research Centre,
Indian Institute of Science, Bangalore, India.
Email: {sree@hpc.serc,govind@serc,mjt@serc}.iisc.ernet.in

***Abstract*—A new breed of processors like the Cell Broadband Engine, the Imagine stream processor and the various GPU processors emphasize data-level parallelism (DLP) and thread-level parallelism (TLP) as opposed to traditional instruction-level parallelism (ILP). This allows them to achieve order-of-magnitude improvements over conventional superscalar processors for many workloads. However, it is unclear as to how much parallelism of these types exists in current programs. Most earlier studies have largely concentrated on the amount of ILP in a program, without differentiating DLP or TLP.**

In this study, we investigate the extent of data-level parallelism available in programs in the MediaBench suite. By packing instructions in a SIMD fashion, we observe reductions of up to 91% (84% on average) in the number of dynamic instructions, indicating a very high degree of DLP in several applications.

I. Introduction

Modern superscalar processors exploit instruction-level parallelism (ILP) in a program to achieve performance. This ILP is a combination of one or more of SISD, SIMD, MISD and MIMD [1] parallelism. As ILP is usually implicit and discovered at runtime, exploiting it has always been complex and challenging, and it has become increasingly difficult to do so in a complexity-effective manner.

Many recent production and research processors have therefore taken the explicit parallelism route, and require the programmer (or compiler) to indicate regions of SIMD or MIMD parallelism. These data-level parallel processors like the Cell Broadband Engine [2], the Imagine Stream Processor [3] and the various GPU processors, have been able to achieve order of magnitude performance improvements over conventional processors at significantly lower hardware complexity for many important workloads.

Distinguishing the contribution of data-level parallelism (DLP) and thread-level parallelism (TLP) to the overall parallelism in a program is therefore necessary. In this study, we examine a number of programs from the MediaBench suite to determine the degree of data-level parallelism present. By packing the instructions in these programs in a SIMD fashion, we find an average of 84% and upto a maximum of 91% reduction in the number of dynamic instructions in these programs, indicating a DLP of 4 or more.

II. Related Work

Earlier limits studies, like [4] and [5], have investigated the degree of ILP in programs. However, they have not differentiated their results into DLP, TLP or the like. Our study uses similar algorithms and techniques to uncover parallelism but reports the degree of SIMD parallelism (a form of DLP) in a program. This differentiation is important because SIMD DLP is often far easier to exploit than non-SIMD ILP.

Scott and Davidson attempted to explore the limits of SIMD parallelism and evaluated Spec95 integer and MediaBench benchmarks [6]. Their results for the MediaBench programs were uncharacteristically pessimistic. Our study explores only the MediaBench suite, but differs from [6] in the following respects:

1) We demonstrate that the earlier approach unnecessarily constrained the DLP by considering instructions in program order. By relaxing this, and by considering a topological sort of the instructions in the dynamic data dependence graph, we eliminate packing limitations and show significant reduction in the number of packed instructions.
2) We establish an upper limit on the parallelism that any algorithm seeking to pack instructions in a SIMD fashion can achieve.
3) We recognize pseudo-dependences that are an artefact of compilation (e.g., iterator dependences [5]), and which tend to artificially limit available parallelism.

III. Experimental Methodology

Our packing algorithm operates on the dynamic trace of a program. We annotate this trace with true data dependence information obtained by doing perfect register and memory disambiguation. Control dependences are ignored since we do perfect branch prediction.

A. Packing Algorithm

Our packing algorithm (Algorithm 1) is a variant of the algorithms used in other limit studies [4], modified to handle SIMD semantics. It transforms an input trace of dynamic instructions into a sequence of instruction-containing *packs* that satisfy the following constraints:

- A pack only contains instructions that have the same opcode.
- All instructions in a pack are independent of each other.
- Instructions in pack i may depend only on instructions in pack 1 to pack $i-1$.
- Instructions that modify control flow do not pack with any instruction.

We do not constrain the width of a pack, nor the widths of operands that a pack contains, as our intention is to discover the limits of DLP.

Algorithm 1 The Packing Algorithm

```
for all instructions Insn in Unpacked do
  LastSource ← Source of Insn in PackingWindow
      which occurs last (if any)
  if LastSource exists then
    SearchRange ← [NEXTPACK(LastSource),
                   LASTPACK(PackingWindow)]
  else
    SearchRange ← [FIRSTPACK(PackingWindow),
                   LASTPACK(PackingWindow)]
  end if
  CompatPack ← PACKWITHSAMEOP(SearchRange,
                              Insn)
  if CompatPack found then
    Add Insn to CompatPack
  else
    NewPack ← {Insn}
    if LastSource exists then
      Insn NewPack after LastSource
    else
      Insn NewPack at bottom of PackingWindow
    end if
  end if
end for
```

Although the proposed algorithm can operate on the entire program trace *in toto*, detecting dependences across the entire program significantly slows it down. Therefore we operate the algorithm in a sliding window fashion. A $PackingWindow$ is used to accumulate packs as they are created. When a newly inserted pack causes the length of the $PackingWindow$ to exceed a certain limit, we slide off the pack at the top of the $PackingWindow$.

To increase opportunity for packing, we also perform the following transformations on the input trace:

1) Topological Sort: The instructions are fetched into a reorder window where they are topologically sorted based on their dependence graph. The size of the reorder window is the same as the size of the $PackingWindow$. All instructions in this window are then packed before the window is refilled and the procedure repeated.

Since, for an instruction, the range searched for compatible packs is from the source to the end of the $PackingWindow$ (Algorithm 1), this transformation increases the number of packs that are searched.

2) Iterator Dependences: As pointed out in [5], iterator dependences are pseudo-dependences that can severely limit available parallelism. These are operations like `i++` that introduce inter-iteration dependences, but are actually loop maintenance instructions and of no importance to the actual algorithm. We handle them in two ways. Firstly, we use aggressive loop unrolling at the compiler level, which eliminates many of these instructions from the dynamic trace. Secondly, we identify and modify the dependence graph for those instructions that still remain in the dynamic trace.

Identification is done in two steps. In the first step, we mark all those instructions that are self-dependent (i.e., depend on a dynamic instruction having the same PC as themselves). Further, since loop unrolling may create multiple static copies of the same instruction (i.e., different PC), we also mark instructions whose source instructions have exactly the same form as themselves. The second step checks if one of the source operands of a marked instruction is an immediate operand. If it is, then the instruction is an iterator dependence and the dependence is relaxed. Otherwise we assume it is a reduction, e.g., `sum = sum + a[i]`, and the dependence is honoured.

Once an iterator dependence has been identified, it is packed with its source instruction, if the source exists in the $PackingWindow$.

B. Benchmark Programs

For this study, we use the programs shown in Table I from the MediaBench suite of benchmarks. These benchmarks were compiled for the Alpha ISA using gcc 3.4.6 with the options `-O2 -funroll-loops -floop-optimize --param max-unroll-times=128`

We modified SimpleScalar 3.0's [7] `sim-safe` to emit dynamic instruction traces. All programs were run to completion.

Benchmark	Trace Lengths
JPEG	`jpegencode` (18M), `jpegdecode` (6M)
MPEG2	`mpeg2decode` (185M), `mpeg2encode` (1.8B)
GSM	`gsmencode` (339M), `gsmdecode` (101M)
EPIC	`epic` (56M), `unepic` (11M)

TABLE I
MEDIABENCH PROGRAMS STUDIED AND THEIR TRACE LENGTHS IN INSTRUCTIONS

C. Output Instruction Set

To explore data-level parallelism in a more complete manner, we do not pre-determine the output SIMD instructions. Instead, we enforce the following SIMD semantics:

- All instructions in the pack have the same opcode.
- All instructions in the pack are independent of each other.
- Control instructions do not form packs.

Thus, any instruction that exhibits SIMD behaviour will form packs. This includes those instructions that have conventional SIMD equivalents (e.g., arithmetic and logical instructions) as well as those that may not. We do not attempt to identify complicated SIMD operations like sum of differences, etc. that many SIMD instruction sets also provide.

D. Performance Metric

We report the available DLP in a program by measuring the reduction in the number of dynamic instruction in the packed form. That is:

$$Reduction \quad = \quad 1 - \frac{NumOutputPacks}{NumInputInsns}$$

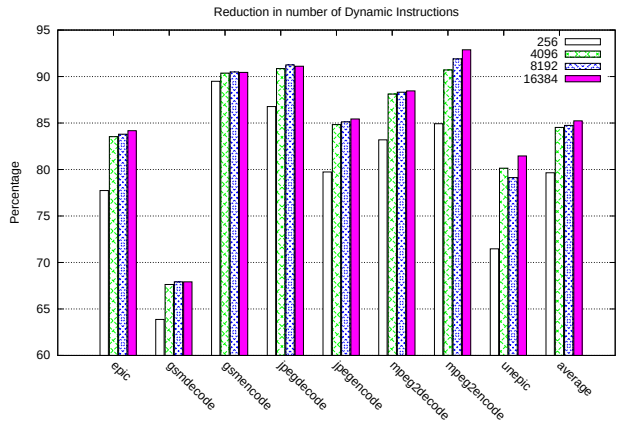

Fig. 1. Reduction in dynamic instructions for different window sizes

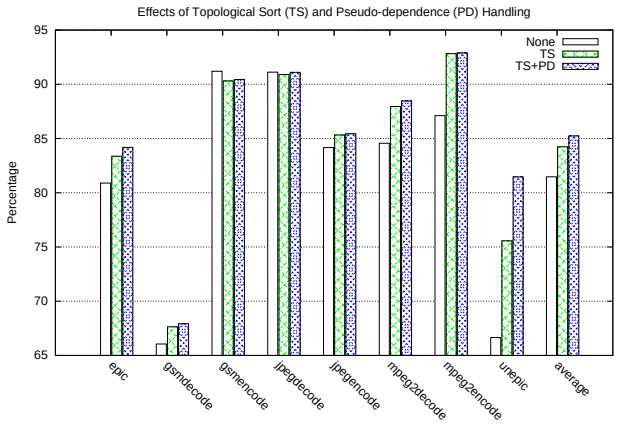

Fig. 2. Effects of Topological Sort and Pseudo-dependence handling (PackingWindow size is 16384)

E. Establishing the Perfect Limit

To establish a lower bound on the SIMD packing attainable by *any* algorithm, we observe that if all instructions in the trace are independent:

$$NumOutputPacks_{perfect} = NumControlInsns + NumDistinctOpcodes$$

and

$$Reduction_{perfect} = 1 - \frac{NumOutputPacks_{perfect}}{NumInputInsns}$$

Note that this is a lower bound as no algorithm can produce fewer packs. However, for very large windows, using the critical path length in place of the number of distinct opcodes will possibly yield a more realistic number. For our small windows, this metric is adequate.

Since our algorithm works on a per-window basis, $Reduction$ and $Reduction_{perfect}$ are calculated every x input instructions, where x is the size of $PackingWindow$. This means that, in some cases, values of $Reduction$ may exceed $Reduction_{perfect}$ because input instructions in the current input window will join packs formed by instructions in the previous input window and not form any new packs.

IV. Results

As Figure 1 shows, we achieve a reduction in the number of dynamic instructions by 84% on average and upto 90% for `jpegencode`, `gsmencode` and `mpeg2encode`. This translates to an average DLP of 4 and up to nearly 5 instructions per pack.

The effects of topological sorting and pseudo-dependence handling vary by program. As shown in Figure 2, programs like `epic`, `unepic` and `mpeg2decode` get pronounced increases in dynamic instruction reduction due to topological sorting. However, only `unepic` seems to benefit significantly from handling of pseudo-dependences. We observe higher reductions in our approach than the 25% observed in [6] for these same programs. We attribute this to the following reasons:

1) We use a continuous sliding window as opposed to a discrete window for packing.
2) We do not limit the size of a pack.

Since real machines impose various limitations on the form of SIMD packs, we investigate the effect of the following constraints on pack formation (and hence reduction achieved):

1) MEM-CONTIGUOUS: Load/store instructions in a pack must address only contiguous regions of memory.
2) PACKSOME: Only a limited set of arithmetic and logical instructions are allowed to form packs. Without this constraint, any instruction exhibiting SIMD behaviour would have been allowed to form packs.
3) WIDTH-x: Packs formed can contain a maximum of x operations. We assume all operations to be of the same size, a reasonable assumption given that a majority of the Alpha instruction set operates on 64-bit operands. We choose x to be 4 or 8.
4) WIDTH-x-PACKSOME: Pack formation must obey the constraints of both PACKSOME and WIDTH-x.

Figure 3 shows the reductions achieved under these constraints for a $PackingWindow$ of size 16384 and with topological sort applied. Constraining the pack width leads to average drops of 20% for a width of 4, and a 10% for a pack width of 8. The effects of PACKSOME are also significant, for about half of the benchmarks, it causes a drop in reduction that is greater than WIDTH-4. On average, its drop is about 20%, nearly the same as WIDTH-4. Enforcing memory contiguity results in an average drop of 13%. The WIDTH-x-PACKSOME constraint which enforces two of the constraints at the same time results in an average drop of 27%, slightly less than the

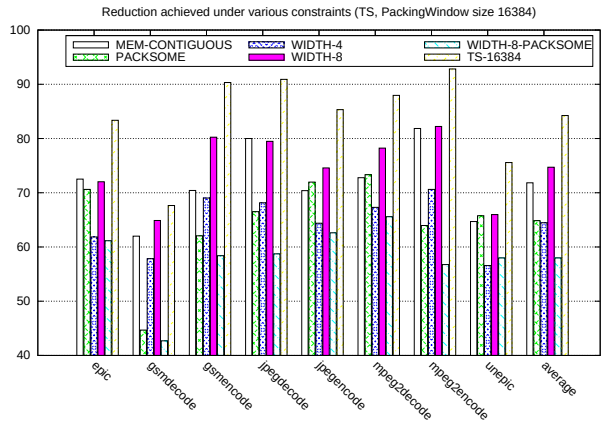

Fig. 3. Achieved reduction when several constraints are imposed on packing; MEM-CONTIGUOUS=Packs containing memory operations only address contiguous memory, PACKSOME=Only some instructions can form packs, WIDTH-x=Maximum x operations per pack (TS, PackingWindow size is 16384)

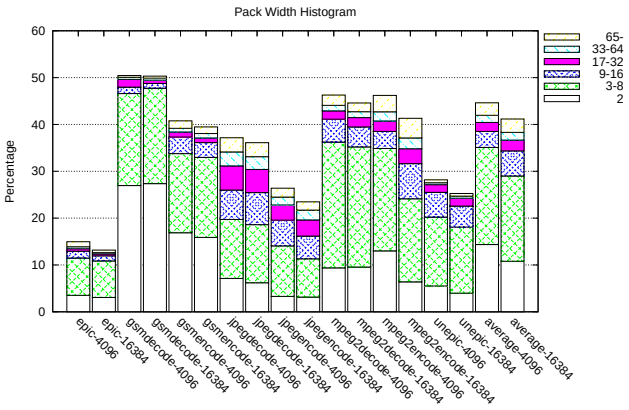

Fig. 4. Widths of generated packs

sum of the drops due to the individual constraints.

Figure 5 shows that, on average, 90% of instructions in the dynamic trace participate in the formation of packs of width 2 or more. These packs form about 40% of the packed output (Figure 4).

The values of $Reduction_{perfect}$ and $Reduction$ for `unepic` are plotted on a per-window basis in Figure 6 for a window size of 16384. Variations in data-level parallelism by program region are clearly visible. The two horizontal lines running across the length of the graph denote the averages of these two values respectively. We observe that the average value of $Reduction$ is within 10% of $Reduction_{perfect}$.

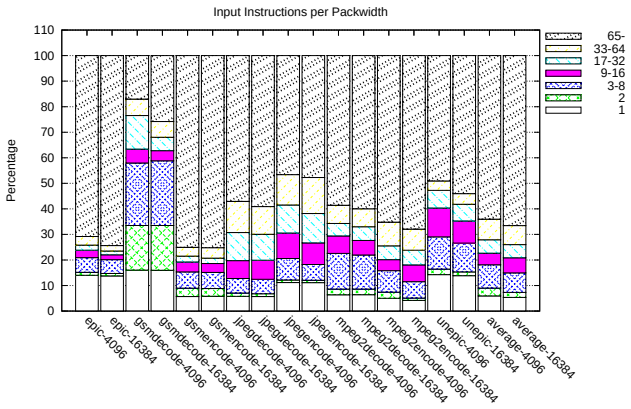

Fig. 5. Input instructions per packwidth

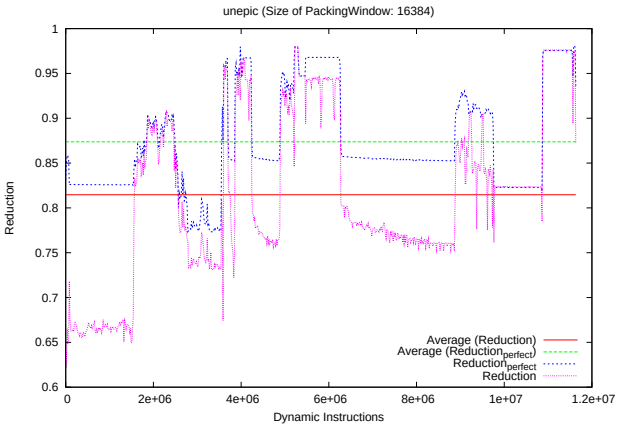

Fig. 6. Achieved vs perfect reduction (TS+PD) on a per-window basis for `unepic` when PackingWindow is of size 16384

V. Conclusion

Most modern superscalar processors have gained SIMD execution capabilities in the last decade, and these have been used to great effect to accelerate multimedia and other data-parallel workloads. Indeed, recent processors like the Cell Processor [2] are designed primarily to support SIMD execution.

In our study, we determined the amount of data-level parallelism in benchmarks from the MediaBench suite. We developed algorithms that are sensitive to the packing limitations imposed by SIMD semantics, in contrast to earlier studies.

Our results show that, on average, over 90% of dynamic instructions participate in the formation of packs of width 2 or more. These packs consist 40% of the packed output. Indeed, about 85% of dynamic instructions get packed into packs of width 3 or more, forming 20% of the packed output. Dynamic instruction trace lengths reduce by 84% on average, and up to

91% for some programs indicating a DLP of 4 or higher.

REFERENCES

[1] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 0018-9219, pp. 1901–1909, 1966.

[2] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, 2005.

[3] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The Imagine stream processor," *International Conference on Computer Design*, 2002.

[4] D. W. Wall, "Limits of instruction-level parallelism," Digital Western Research Laboratory, Tech. Rep. 93/6, Nov. 1993.

[5] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, 1992.

[6] K. Scott and J. Davidson, "Exploring the limits of sub-word level parallelism," in *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[7] D. C. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," Computer Sciences Department, University of Wisconsin–Madison, Tech. Rep. 1342, May 1997.