

Estimating Potential Parallelism for Platform Retargeting

Linda Wills, Tarek Taha, Lewis Baumstark, Jr., Scott Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332-0250
{linda.wills, taha, lewisb, scott.wills}@ece.gatech.edu

Abstract

Scientific, symbolic, and multimedia applications present diverse computing workloads with different types of inherent parallelism. Tomorrow's processors will employ varying combinations of parallel execution mechanisms to efficiently harness this parallelism. The explosion of consumer products that incorporate high performance embedded computing will increase the stratification of the processor design space. However, existing code assets are limited to sequential expression of what should be highly parallel algorithms. Retargeting to parallel mechanisms is difficult, but can provide significant increases in efficiency. It is desirable to estimate potential parallelism before undertaking the expensive process of reverse engineering and retargeting. This paper presents a lightweight dynamic analysis technique for characterizing the types of parallelism that are inherent in a given program to estimate the potential benefit of retargeting. The technique is validated on Spec95 and MediaBench benchmarks widely used to evaluate processor performance. Results correlate well with previous experience in parallelizing these well-understood applications.

1. Introduction

Growing demand for portable multimedia products (e.g., digital cameras, wireless PDAs, and digital video recorders) is driving the development of high performance, high efficiency processors. Consumer product production volumes allow processor designs that employ specialized parallel computing mechanisms. Meanwhile, general-purpose microprocessors are nearing the limit of what can be achieved with instruction parallelism mechanisms (ILP) such as pipelining and superscalar execution [1]. New data level parallelism (DLP) and thread level parallelism (TLP) are being examined for continued increases in processor performance. Their effectiveness depends on their ability to exploit the inherent parallelism in the target applications. For example, image processing algorithms contain

significant data parallelism since operations can be applied concurrently to each pixel of a large image. This has motivated multimedia instruction set extensions (e.g., Intel's MMX [2]). As the complexity of traditional instruction level parallelism mechanisms grows, much attention has been focused on thread level parallelism mechanisms such as simultaneous multithreading [3,4], and chip scale multiprocessors [5].

The primary stumbling block in the transition from ILP-only processors to hybrid parallelism processors is software compatibility. Widely used programming languages (e.g., C, C++) lack the expressiveness to represent explicit parallelism. Processor designers have spent 35 years creating clever methods to execute inherently sequential machine instructions in parallel (a modern microprocessor is capable of concurrently executing over 40 instructions). But few automatic retargeting tools exist for data and task parallelism. Since companies are reluctant to abandon significant sequential code assets, programs must be manually reverse engineered and retargeted for explicitly parallel execution mechanisms. This process is time consuming and expensive; and potential performance improvements are difficult to estimate until lengthy, often undocumented programs are reverse engineered.

The performance improvement estimation is further complicated by the vast design space of hybrid parallel execution machines. There is also a tremendous variation in the hardware efficiency of parallel execution mechanisms. ILP mechanisms can exploit data parallelism; but it does so much less efficiently than DLP mechanisms. In the Pentium II, support for 4-way instruction level parallelism increased chip area by over 40%, while support for 8-way data parallelism increased chip area by less than 5%. Determining which types of parallelism are inherent in a given program and retargeting the program to the hardware architecture with the capacity to efficiently exploit it is critical to meeting the computational demands of the application.

The reverse engineering and retargeting process would significantly benefit from an automatic program analysis tool that estimates the types and

amounts of parallelism present in an existing program. Analogous goals have been pursued by Burd and Munro [6] in predicting potential reuse candidates as a preliminary stage to guiding reengineering of legacy code.

This paper presents a lightweight technique for reverse engineering the types and amounts of parallelism inherent in a program to estimate the potential benefit of retargeting to a hybrid parallel mechanism processor. This technique suggests the type of parallelization strategy to use in retargeting and helps identify requirements for the processor that best exploits the parallelism. This is especially important for new reconfigurable processors that can be tailored to specific applications. An architecture employing appropriate parallel execution mechanisms can be specified based on the needs of the application and the types of parallelism that should be exploited. Thus, this technique helps with architecture exploration as well as guiding the program retargeting.

This paper first gives background on the basic types of parallelism and a model of how they are related to each other. It then presents a dynamic analysis approach to estimating parallelism parameters for a given program. A validation experiment is then presented in which a set of benchmark programs from the SPEC95 [7] and MediaBench [8] suites are analyzed to estimate their parallelism potential. These estimates correlate well with understanding of these widely used programs.

2. Taxonomy of parallelism

This technique is based on a generic architecture model developed by [9] that categorizes architectural organizations into three broad classes, depending on the type of parallelism exploited. The bottom level is the *data parallel* class in which parallelism is achieved by broadcasting, at each step in the execution, one operation to a set of N processing elements. For example, if an array of processors is being used for image processing and each processor held one pixel of an image, a thresholding operation can be applied to each pixel value to convert the image from gray scale to black and white. This is the type of parallelism exploited in single-instruction-multiple-data machines (SIMD) dating back to the 1960's (e.g., ILLIAC-IV, MPP, TMC CM, Maspar). More recently, it is exploited by the new subword parallelism extensions to traditional instruction sets, such as MMX and AltiVec [2], [10-13]. These extensions allow multiple operands (e.g., pixel values)

to be packed into a single data word so that a single operation can be applied to each subword in parallel (e.g., adding two packed eight operand data words simultaneously produces eight additions).

The next level in the taxonomy is the *instruction level parallel* (ILP) class of organizations in which one thread of control schedules multiple (usually different) operations for execution on different processing units. For example, a superscalar or VLIW processor might have two integer ALU functional units, a floating point multiply unit, and a floating point divide unit. The code for this processor might be scheduled so that an integer add, a floating point multiply, and floating point divide instruction are all executed at the same time, if the instructions are independent (i.e., have no true data dependences between them).

Data parallelism is a proper subset of instruction level parallelism. An ILP processor can exploit data parallelism if it has multiple versions of each type of functional unit. It simply schedules multiple instructions of the same type on multiple functional units of the same type. On the other hand, instruction level parallelism cannot be exploited by a data parallel machine organization. Instruction level parallelism is exploited by concurrently executing different instructions on different functional units. A program that has independent instructions (i.e., instructions that have few data dependences between them) that involve different types of operations has a great deal of inherent instruction level parallelism, but no data parallelism. In data parallel architectures, a common instruction is broadcast to all processing elements, each of which has its own data. Such an architecture would not be able to execute the independent instructions in parallel because it can only broadcast one type of instruction at a time.

The third level in the parallelism taxonomy is *thread-level parallelism* (TLP). In this class of architectures, multiple threads of control are executed, each of which involves multiple operations that may be executed on multiple functional units. Simultaneous multithreading (SMT) and chip scale multiprocessors (CMP) exploit this type of parallelism.

Instruction level parallelism is a proper subset of thread level parallelism. An ILP program can be executed on a machine designed to exploit TLP by creating multiple copies of the ILP's thread of control. The reverse is not true. The TLP inherent in a program having multiple independent threads of control cannot be exploited by an ILP machine organization, since it can only execute one thread of control at a time.

Given the relationships among primary types of parallelism, [9] developed a workload model to characterize the parallelism inherent in a given application workload. This is shown in Figure 1. The model starts with Amdahl's law that says that any workload consists of a portion that can only be executed serially (*serial_fraction*) and a portion that can be executed in parallel (*parallel_fraction*). The model breaks the parallel fraction into the fraction of parallelism that can only be exploited by a TLP architecture (*TLP_fraction*), the fraction that can only be exploited by TLP or ILP (*ILP_fraction*), and the fraction that can be exploited by any of the three types of parallelism (*DLP_fraction*).

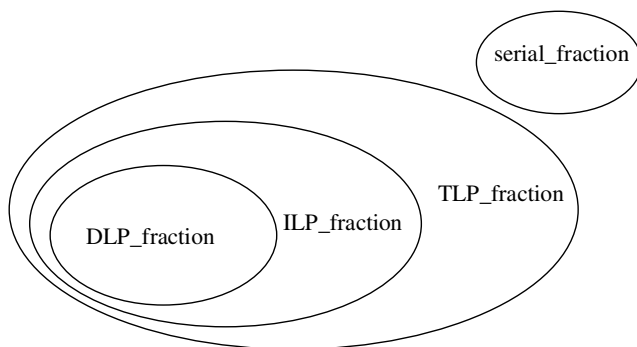


Figure 1: Relationship of parallelism types.

The workload model uses these parameters to generically describe the parallelism present in a given application. Such a model for an application can be used to guide its retargeting to best extract its parallelism and to choose an architecture that has the capacity to exploit that parallelism. This paper presents a technique for automatically estimating these workload parameters for an arbitrary program.

3. Overview of analysis technique

The approach to estimating workload parameters for a given program is to perform a dynamic analysis of the program to detect opportunities for instructions to be executed in parallel. It consists of three main steps as shown in Figure 2. First the execution of the program is simulated to produce a sequential instruction trace. This trace is then fed into a scheduler that extracts the maximum available parallelism by detecting opportunities for independent instructions to be scheduled to execute concurrently. The scheduler is able to impose different constraints with regard to control dependencies that model different types of parallelism support. The schedule is then analyzed based on the number of instructions scheduled in the same time slot and on the frequency

profile of repeatedly occurring instructions. Each of these steps is described in this section.

3.1 Generating a dynamic instruction trace

The simulator used in this study is SimpleScalar [14], which is an open-source simulator widely used in computer architecture research. The simulator takes as input programs that were compiled (e.g., using gcc) to SimpleScalar's PISA instruction set. PISA stands for Portable Instruction Set Architecture and it encompasses features of most common RISC instruction sets currently in use.

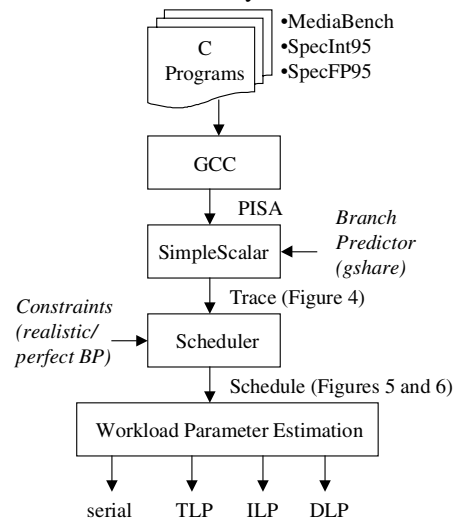


Figure 2: Parallelism Estimation Technique

The simulator generates a trace of the actual instructions that are executed for a given program applied to a specific input data set (e.g., see Figure 3). The simulator models the computational behavior of the processor in that it computes the results of instruction operations and models the storage and retrieval of results from registers and memory. Single stage instruction execution is assumed. The simulator computes the actual memory locations accessed (read/written) and it resolves branch predicates to determine the direction branches actually take. This reveals the true data dependences and control flow behavior of the program for a given input data set. This information is fed into the scheduler (as described in the next section) to determine which instructions are independent and can be scheduled to execute concurrently.

Memory address disambiguation and branch predicate resolution are critical to this analysis, which is the primary reason dynamic analysis is used as opposed to static analysis. It is difficult and not always possible with static analysis to accurately determine

memory addresses and to resolve branch predicates, particularly for pointer-based accesses and for data-driven branches. On the other hand, a potential drawback with dynamic analysis is that the results obtained are dependent on the input data set. However, this technique is intended to be applied to existing code for which there are established typical workload data sets.

1	lw \$1, 100(\$0)
2	lw \$2, 200(\$0)
3	sub \$3, \$1, \$2
4	beq \$3, \$2, target
5	andi \$4, \$2, 0xFF00
6	mul \$5, \$4, \$4
7	lw \$4, 300(\$0)
8	sll \$3, \$5, 3
9	add \$3, \$3, \$5
10	bne \$4, \$0, loop2
11	lw \$6, 400(\$0)
12	slli \$7, \$6, 0
13	andi \$6, \$6, 0xFF
14	mul \$7, \$7, \$6

Figure 3: An example program trace. Instruction 4 is a branch that was predicted correctly. Instruction 10 is a mispredicted branch.

Since we are interested in the upper limit of parallelism available in the program, independent of resource limits of any particular hardware architecture, we simulate program execution under perfect resource assumptions. This means there are no limits placed on available architectural resources, such as the following:

- the number of functional units (e.g., arithmetic-logical units, multipliers, floating point divide units),
- the amount of memory (e.g., no cache misses or page faults are simulated), or
- the number of registers (perfect register renaming is performed to eliminate any anti- or output-dependences).

Single cycle instruction execution is also assumed because we are not interested in modeling variable execution latencies for a particular hardware architecture, but rather in estimating an upper limit on the potential for parallelism of different types.

The simulator models state-of-the-art dynamic branch prediction schemes. Dynamic branch prediction is a hardware technique that keeps track of information about how branches behaved in the past and uses this history to predict whether or not a branch will be taken in the future. For example, some predictors work by keeping track of what happened

the last couple of times a branch was encountered and if it detects that a branch is biased in a certain direction, it will predict that it will follow that bias. Loop branches, for example, are typically biased toward branching back to the start of the loop body. When a prediction is made, the hardware can start “speculatively” executing instructions along the predicted path even before the branch predicate has been resolved. If the prediction was correct, the processor has gotten a head start on executing the instructions.

In our study, SimpleScalar was used with the gshare branch prediction scheme. The branch prediction strategy used does not affect the dynamic instruction trace generated by SimpleScalar. Only the “correct” instructions occur in the trace; no instructions along a mispredicted path are included. The branch prediction scheme is simulated to generate a prediction for each branch and to record which branches would be mispredicted by a real machine using that type of prediction scheme. Simulating branch prediction allows us to identify which branch instructions in the program tend to be unpredictable.

The ability to predict branches reliably is important in exploiting parallelism because it allows instructions after a branch to be executed before the branch is actually completed. Previous studies [15,16] have shown that available parallelism within a basic block is severely limited. Speculative execution is widely used to allow instructions to cross branch boundaries and to allow more instructions to execute concurrently. Mispredictions on the other hand, cause wasted work and a branch misprediction penalty in that the correct instructions can only begin once the branch predicate is resolved and the misprediction is detected.

3.2 Scheduler

The scheduler takes the trace of instructions and schedules each instruction in turn into time slots as early as possible while still satisfying data dependence constraints. There is no limit on the number of instructions that can be scheduled in a time slot. The goal is to extract parallelism by bringing instructions together to execute concurrently that may be separated by great distances in the lexical program order.

The ability to schedule across branch boundaries is critical to finding multiple instructions to execute concurrently. The scheduler creates two different schedules: one that enforces constraints on how instructions are scheduled relative to mispredicted branches and the other that enforces no constraints

due to control flow (i.e., effectively assuming perfect branch prediction). These two schedules are useful in studying the amount of ILP and TLP in a given program. It is well known that unpredictable branch behavior limits the ability to exploit ILP. TLP, on the other hand, involves speculatively executing multiple control paths, moving beyond unpredictable branches so that mispredictions are not a factor. More precisely, all schedules obey the following constraint:

An instruction A must be scheduled after another B if there is a true data dependence between A and B (i.e., B's result is one of A's operands).

Other types of dependences, namely anti-dependences (write-after-read) and output dependences (write-after-write), do not occur among the instructions in the trace because the simulator performs perfect register renaming.

Additionally, the two types of schedules are generated by either:

- imposing no additional constraints (perfect branch prediction – PBP), or
- imposing the realistic branch prediction (RBP) constraint: if a mispredicted branch B occurs before an instruction C in the trace, then C cannot be scheduled before B. (However, if B is predicted correctly, any instruction following it in the trace can be scheduled before B provided data dependences are satisfied.)

Inst#	Cycle				
	0	1	2	3	4
1	lw				
2	lw				
3		sub			
4			beq		
5		andi			
6			mul		
7	lw				
8				sll	
9					add
10		bne			
11			lw		
12				slti	
13				andi	
14					mul

Figure 4: Schedule generated under realistic branch prediction constraints.

Figure 4 shows the schedule for the trace in Figure 3, generated under realistic branch prediction constraints. Note that instruction 5 is scheduled before instruction 4 (which is a correctly predicted branch). Also, instructions 11-14 must be scheduled after instruction 10 (which is mispredicted).

Figure 5 shows the schedule for the trace in Figure 3, generated under perfect branch prediction constraints. Note that instructions 11-13 are not restricted to occurring after instruction 10, revealing opportunities for more instructions to be executed in parallel in the same slot.

3.3 Scheduler implementation using limit window

The technique we use to estimate parallelism from the schedules is a variation of the conventional “limit window” analysis technique [17,18], extended to include analysis of potential data parallelism in addition to ILP and TLP estimates. Each instruction in the dynamic instruction trace is scheduled into timeslots within windows that impose constraints on the earliest time at which an instruction can begin executing. Different scheduling constraints model the effect of different combinations of parallelism support and also manage the tractability of the scheduling over very large traces.

Inst#	Cycle				
	0	1	2	3	4
1	lw				
2	lw				
3		sub			
4			beq		
5		andi			
6			mul		
7	lw				
8				sll	
9					add
10		bne			
11	lw				
12		slti			
13		andi			
14			mul		

Figure 5. Schedule generated under perfect branch prediction constraints.

To model the realistic branch prediction (RBP) constraint, all instructions up to a mispredicted branch are scheduled in an instruction window. Their relative ordering is determined by data dependences. Upon encountering a mispredicted branch, all slots up to and including the one containing the mispredicted branch are retired, which means no more instructions may be added to those time slots. The instruction window is then moved past the mispredicted branch instruction's time slot so that all future instructions are scheduled past the mispredicted branch.

Instructions that are before a mispredicted branch in a trace might be scheduled to execute with instructions after the branch, but no instructions after a mispredicted branch can ever be scheduled to execute with instructions before the branch. This is illustrated in Figure 4.

To model perfect branch prediction (PBP), all instructions should be scheduled in a virtually unlimited size instruction window. There are no mispredicted branches to create boundaries between successive windows. However, we are implementing this on a finite machine and need to set a limit on the instruction window size for tractability of scheduling; 4096 timeslots was chosen as the maximum size for an instruction window.

With perfect branch prediction, it is possible that a data dependence chain of 4096 instructions (the default limit on the instruction window size) will eventually form and instructions will have to be scheduled beyond the finite number of slots in the window. To create a virtually unlimited window size, when 4096 time slots are filled by a dependence chain, a portion of the window is retired (specifically, the first half), and the window is moved forward past all retired slots.

By retiring only a portion of the window, there is an overlap in the slots between the old and new window positions. This allows the machine to mimic an almost continuous window throughout the execution of a program. Although instructions can no longer be scheduled into retired slots, it is assumed that a realistic machine would be very unlikely to schedule instructions very far apart (in this case over at least 4096 instructions apart into the same slot.

3.4 Estimators

Our use of the limit window analysis technique differs from how it has been used in the past. Previous work [17,18] attempts to measure upper limits on parallelism by computing the speedup in total execution time of a scheduled trace (with operations occurring in parallel) to that of the original sequential trace. The execution time corresponds to the time at which the last instruction in the schedule completes. Parallelism is measured as the ratio of the sequential trace's execution time to the parallel schedule's execution time. This captures the combined effect of different types of parallelism on the overall execution time of a given application. However, it does not give insights into the relative amounts of different types of parallelism that contribute to producing this overall effect.

Our variation of the limit window analysis differs from the conventional in how measures of parallelism are computed from the scheduled trace. The time slots in each schedule are examined to extract operation type frequency measures that determine, for example, how often the same operation can be concurrently executed or how many different operations can be executed at the same time.

This section defines how each of the parameters in the workload model is estimated.

Serial Fraction

The serial fraction corresponds to the fraction of instructions that must be executed sequentially in a given application. If a program of 1 million instructions executes in 1 million cycles, then all the instructions are performed sequentially and the serial fraction is 1. If the same program executes in 250,000 cycles (or at a rate of 4 instructions per cycle), then a quarter of the instructions execute sequentially and three fourths execute in parallel, so the serial fraction is 0.25 and the parallel fraction is 0.75.

In general, the serial fraction is the inverse of the average number of instructions executed per cycle (IPC) obtained with perfect branch prediction conditions in the scheduling of instructions into time slots:

$$\text{Serial_fraction} = 1/\text{IPC}_{\text{pbp}}$$

$$\text{IPC}_{\text{pbp}} = \frac{1}{T} \sum_{i=1}^T I_{ip}$$

where T = number of time slots,

I_{ip} = total number of instructions in time slot i with *perfect* branch prediction.

Thread Level Parallelism

The TLP_fraction is the complement of the serial_fraction, since it represents the maximum fraction of parallelism that can be obtained (by relaxing control dependencies on mispredicted branches).

$$\begin{aligned} \text{TLP_fraction} &= 1 - \text{Serial_fraction} \\ &= (1 - 1/\text{IPC}_{\text{pbp}}) \end{aligned}$$

Instruction Level Parallelism

ILP_fraction is computed similarly as a function of average number of instructions per time slot, but with realistic branch prediction scheduling constraints.

$$\text{ILP_fraction} = (1 - 1/\text{IPC}_{\text{rbp}})$$

$$\text{IPC}_{\text{rbp}} = \frac{1}{T} \sum_{i=1}^T I_{ir}$$

where T = number of time slots,

I_{ir} = total number of instructions in time slot i with *realistic* branch prediction.

Data Level Parallelism

The DLP fraction is measured as the fraction of instructions in each time slot that occurs multiple times. Only multiple instructions with identical program counter (PC) values (i.e., instruction memory addresses) are included. This occurs when instructions from multiple loop iterations are not blocked from the same time slot by true dependences. The DLP factor is computed as the fraction of all instructions that are DLP:

$$\text{DLP_factor} = \frac{\sum_{i=1}^T D_i}{I}$$

where T = number of time slots,

D_i = number of instructions with the identical PC address in time slot i , and

I = total number of instructions in the trace.

This measure is calculated under perfect branch prediction in order to capture full potential data parallelism. Realistic branch prediction would mask possible data parallelism in low control parallelism (poor branch prediction accuracy) environments. So DLP_fraction is defined as:

$$\text{DLP_fraction} = \text{TLP_fraction} * \text{DLP_factor}.$$

4. Validation and analysis

A selection of well-understood programs from MediaBench [8] and SPEC95 [7] benchmarks are analyzed to validate this technique. Using knowledge of the program's purpose, along with its generally accepted characteristics (e.g., image processing programs exhibit greater data parallelism than symbolic programs), expected parallelism distributions are compared against analysis results. Table 1 provides a brief description of each program.

The summary results of the analysis are presented in Figure 6. The percentage of DLP, ILP, TLP, and serial instructions are illustrated as a stacked bar. Since, in general, ILP instructions subsume DLP instructions, and TLP instructions subsume ILP and DLP instructions, the top of the ILP and TLP regions reflect the total percentage of ILP and TLP instructions. For example, the first application, cjpeg, has 85% DLP, 90% ILP, and 99% TLP. The remaining serial fraction for cjpeg is 1%. In the program mpeg2encode, the DLP percentage exceeds the ILP percentage by 2% resulting from the program's high DLP factor and low branch prediction accuracy. Although ILP's definition subsumes DLP, this program's characteristics occasionally generate small inconsistencies in the approximation methods.

Table 1: MediaBench and SPECmark descriptions

Suite	Application	Description
MediaBench (multimedia)	cjpeg / djpeg	Lossy image compression coder and decoder
	epic / unepic	Image compression coder / decoder based on wavelet decomposition and run-length/Huffman entropy coding
	mpeg2encode / mpeg2decode	Video compression coder and decoder using a discrete cosine transform
	toast	Fullrate speech transcoder and decoder based on European GSM 06.10 provisional standard
	pegwite / pegwitd	public key encryption and authentication coder and decoder
	rawcaudio / rawdaudio	Speech compression coder and decoder based on adaptive differential pulse code modulation
Spec INT95 (symbolic)	compress	Compresses and uncompresses file in memory
	gcc	GNU C compiler
	go	Artificial Intelligence program that plays the game of Go
	ijpeg	Graphic compression and decompression
	li	Lisp interpreter
	m88ksim	Motorola 88K chip simulator, runs test program
	perl	Manipulate strings (anagrams) and prime numbers in Perl
vortex	A database program	
Spec FP95 (scientific)	applu	Parabolic/elliptic partial differential equations
	fpppp	Quantum chemistry
	swim	Shallow Water Model with 513 x 513 grid
	tomcatv	Mesh generation program
	turb3d	Simulate isotropic, homogeneous turbulence in a cube
wave5	Plasma physics. Electromagnetic particle simulation.	

The ILP and TLP fractions are high (>75% for ILP, >84% for TLP) across all applications, while the DLP fraction fluctuates significantly between applications. Historically, ILP and TLP parallel execution mechanisms offer greater general applicability than DLP mechanisms. Figure 7 provides a simplified summary of DLP fractions across the applications. As expected, MediaBench programs exhibit greater data parallelism than SPECint programs. Nine out of eleven MediaBench programs have greater than 50% DLP while only one out of eight SPECint programs achieves the 50% DLP mark. This illustrates an expected contrast between multimedia and symbolic applications. The one SPECint program that exceeds 50% DLP is jpeg image compression. The low DLP programs in the MediaBench suite (rawcaudio and unepic) contain interloop data dependences that limit DLP as well as ILP and TLP. SPECfp represents scientific applications that heavily exploit array data types. This also promotes high DLP fractions; four out of six SPECfp applications exceed the 50% DLP level.

Figure 8 illustrates the serial remainder after TLP (black) and ILP (gray plus black) instructions have been counted. Programs with large gray regions (e.g., go, mpeg2encode) will benefit from processors with TLP execution mechanisms such as simultaneous multi-threading (SMT) and chip-scale multiprocessing (CMP).

Programs with small gray areas (e.g., vortex, tomcatv) perform well under ILP execution mechanisms due to their predictable branch behavior. Branch prediction statistics are proportional to the ILP-TLP difference; but they are less accurate since they do not capture the mispredicted branch recovery profile used in the reported analysis method.

Scientific applications from SPECfp have the highest ILP and TLP fractions, typically exceeding 95%. As expected, multimedia programs do less well because of their smaller data array size. Symbolic programs have the lowest ILP and TLP fractions because of irregular data access and poor branch predictability.

On average, ILP mechanisms can exploit all but 8.5% of instructions while TLP mechanisms miss only 3.0%. While this is a small percentage improvement, it is extremely significant as Amdahl's law is beginning to limit performance in today's ILP-only microprocessors. Nearly every microprocessor family (Intel IA32 and IA64, AMD, SPARC, Power, and Alpha prior to its cancellation) is planning implementations that employ TLP mechanisms. As system builders begin to employ new TLP processors, information from this analysis technique can identify programs with the greatest opportunity for improvement and therefore promising candidates for retargeting.

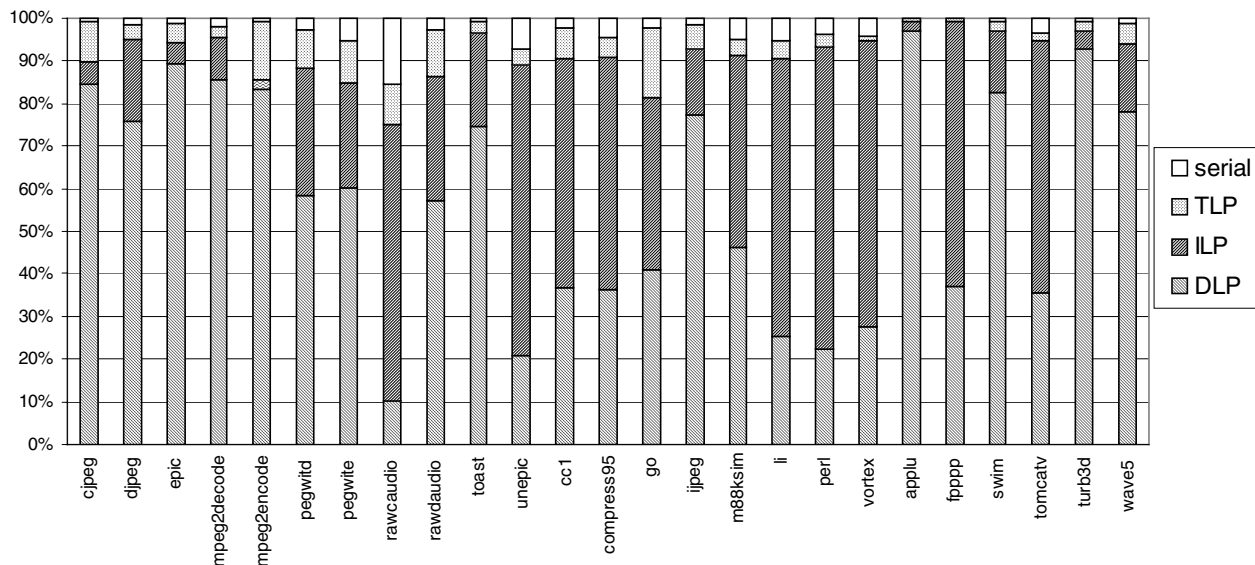


Figure 6: DLP, ILP, TLP, and serial fractions for MediaBench and SPEC applications.

Figure 8: Fraction of serial instructions under LLP (gray) and TLP (black).

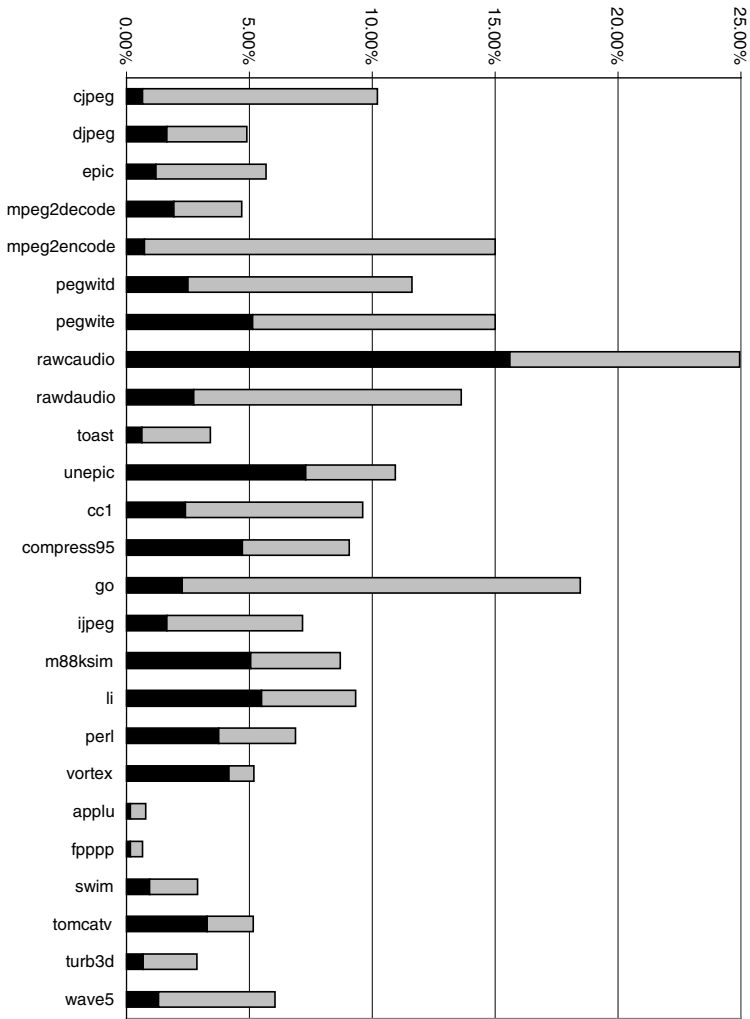
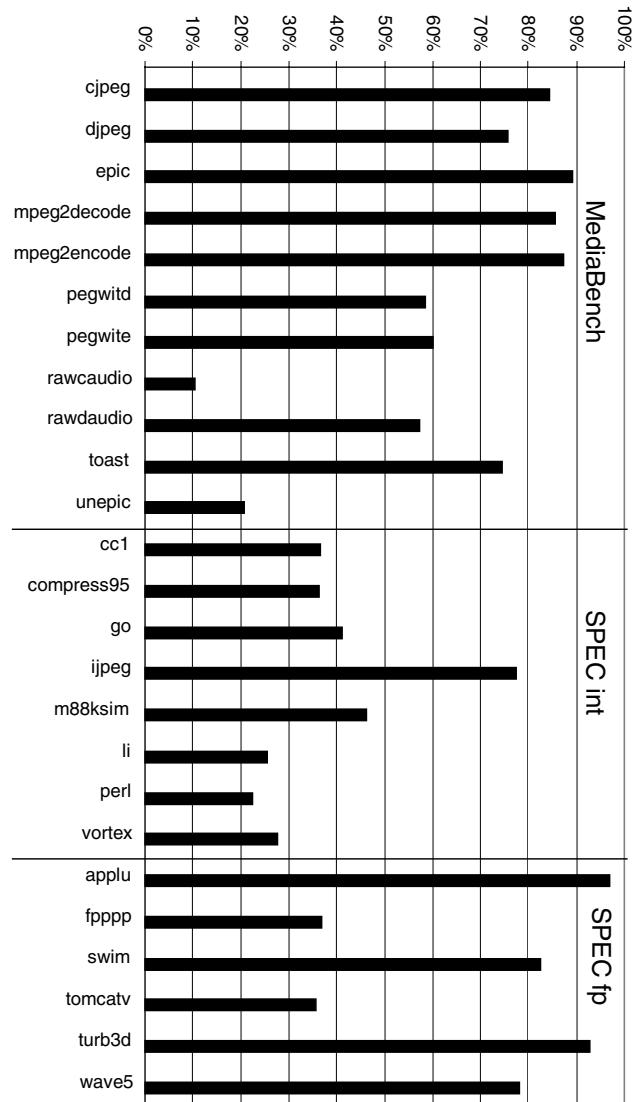


Figure 7: DLP fraction for SPECmark and MediaBench applications



5. Conclusions and future work

A dynamic analysis technique has been presented in this paper for estimating the type and amount of parallelism that is inherent in an unknown program. This technique has been validated by analyzing well-known benchmark suites representing a diverse range of computing workloads. The resulting characterization of their inherent parallelism correlates well with known characteristics of these programs. This increases confidence that the technique will be valuable in guiding efforts to parallelize programs that are less well-understood.

One of the directions for future research is to use models of common machine configurations – specifically, their capacity to exploit various types of parallelism – to provide estimates of the expected benefits of retargeting a given application to a given architecture. For example, architectures can be characterized “generically” using parameters that indicate the machine’s capacity for supporting TLP, ILP, and DLP (e.g., the number of threads of control it can support, the superscalar width, and the number of identical functional units it contains). These parameters can be used in conjunction with the characterization of degree of parallelism in an application program to predict the expected benefit of retargeting that application to a particular machine. This can help not only in choosing the parallelization strategy but in choosing the target architecture as well. Processor synthesis activity is underway at Georgia Tech to generate Verilog descriptions of application-specific processors based on program analysis. Experimental machine specifications will then be programmed into a two megagate FPGA evaluation board, or synthesized into a custom ASIC.

Another area of future research is to experiment with different parameters in the analysis technique, such as different choices of branch predictors to determine their effect on the parallelism estimates.

Finally, the parallelism analysis technique currently is applied to application programs as a whole to give overall estimates of parallelism. It would be beneficial to extend this technique to map the estimates back to particular regions of the program that could benefit from particular types of parallelization (e.g., identify specific loops that seem to have a high potential for data parallelism). This would provide retargeting guidance at a finer level of detail for individual programs to complement the coarser relative characterization across suites of applications.

Acknowledgement

This work was supported in part by the National Science Foundation under NSF CAREER Grant CCR-0092552.

References

- [1] D.W. Wall, “Limits of Instruction-level Parallelism,” in *Proc. ASPLOS-IV*, Santa Clara, CA, pp. 176-188, Apr. 1991.
- [2] A. Peleg and U. Weiser, “MMX Technology Extension to the Intel Architecture,” in *IEEE Micro*, vol. 16, no. 4, pp. 51-59, Aug. 1996.
- [3] D. M. Tullsen, S. J. Eggers, H. M. Levy, “Simultaneous Multithreading: Maximizing on-chip parallelism,” in *Int. Symp. on Computer Architectures (ISCA-22)*, 1995.
- [4] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen, “Simultaneous Multithreading: A Platform for Next-Generation Processors,” *IEEE Micro*, Vol. 17, No. 5, pp. 12-19, Sept/Oct. 1997.
- [5] L. Hammond, B. A. Nayfeh, and K. Olukotun, “A Single-Chip Multiprocessor,” *IEEE Computer*, 30:(9), September 1997, pp. 70-85.
- [6] E. Burd and M. Munro, “Assisting Human Understanding to Aid the Targeting of Necessary Reengineering Work,” in *Proc. 5th Working Conference on Reverse Engineering (WCRE-98)*, pp. 2-9, Honolulu, HI, Oct. 1998.
- [7] Spec CINT95 and CFP95 Benchmarks, Open Systems Group, online at <http://www.specbench.org/osg/>.
- [8] C. Lee, M. Potkonjak, and W. Mangione-Smith, “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems,” *MICRO-30*, www.icsl.ucla.edu/~billms/Publications/mediabench.pdf, 1997.
- [9] L. Codrescu, “ATLAS: A Dynamically Parallelizing Chip-Multiprocessor for Gigascale Integration,” Georgia Institute of Technology, Ph.D. Thesis, April 2000. Available at <http://www.ece.gatech.edu/~scotty/codrescu-thesis.pdf>.
- [10] M. Tremblay, et. al, “VIS Speeds New Media Processing,” *IEEE Micro*, 16:(4), pp. 10-20, Aug. 1996.
- [11] M. Phillip, et al. AltiVec Technology: Accelerating Media Processing Across the Spectrum,” *HOTCHIPS*, Aug. 1998.
- [12] S. Oberman, et al. AMD 3DNow! Technology and the K6-2 Microprocessor, in *HOTCHIPS10*, Aug. 1998.
- [13] R. Lee and M. Smith, Media Processing: A New Design Target, *IEEE Micro*, Vol. 16, No. 4, pp. 6-9, August 1996.
- [14] D. Burger and T. A. Austin. The SimpleScalar tool set, version 2.0. Technical Report #1342. University of Wisconsin-Madison Computer Science, June 1997.
- [15] A. Nicolau and J. Fisher, “Measuring the Parallelism Available for Very Long Instruction Word Architectures,” *IEEE Trans. on Computers*, 33:(11), Nov. 1984, pp. 968-976.
- [16] G. S. Tjaden and M.J. Flynn, “Detection and parallel execution of independent instructions,” *IEEE Trans. on Computers*, vol. C-19, pp. 889-895, Oct. 1970.
- [17] M. S. Lam and R.P. Wilson, “Limits of Control Flow on Parallelism,” *Proc. 19th Int. Symp. on Computer Architecture (ISCA-19)*, pp. 46-57, 1992.
- [18] J. T. Oplinger, D. L. Heine, and M. S. Lam, “In Search of Speculative Thread-Level Parallelism,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, CA, Oct. 1999.