# Limits of Parallelism Using Dynamic Dependency Graphs

Jonathan Mak
University of Cambridge Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue
Cambridge CB3 0FD, United Kingdom
Jonathan.Mak@cl.cam.ac.uk

Alan Mycroft
University of Cambridge Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue
Cambridge CB3 0FD, United Kingdom
Alan.Mycroft@cl.cam.ac.uk

## ABSTRACT

The advance of multi-core processors has led to renewed interest in extracting parallelism from programs. It is sometimes useful to know how much parallelism is exploitable in the limit for general programs, to put into perspective the speedups of various parallelisation techniques. Wall's study [19] was one of the first to examine limits of parallelism in detail. We present an extension of Wall's analysis of limits of parallelism, by constructing *Dynamic Dependency Graphs* from execution traces of a number of benchmark programs, allowing us better visualisation of the types of dependencies which limit parallelism, as well as flexibility in transforming graphs when exploring possible optimisations. Some of the results of Wall and subsequent studies are confirmed, including the fact that average available parallelism is often above 100, but requires effective measures to resolve control dependencies, as well as memory renaming. We also study how certain compiler artifacts affect the limits of parallelism. In particular we show that the use of a *spaghetti stack*, as a technique to implicitly rename stack memory and break chains on true dependencies on the stack pointer, can lead to a doubling of potential parallelism.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*

## General Terms

Languages, Measurement

## Keywords

Limits of parallelism, dynamic dependency graph, spaghetti stack

## 1. INTRODUCTION

The days when rising processor clock speeds translate into automatic performance gains for software is over [18]. In the past software developers had been able to make programs run faster with little effort, simply by waiting for a few years before acquiring a newer and faster computer. Lately, however, physical constraints have meant that single processor clock speeds have levelled off, with manufacturers focusing instead on multi-core technologies. As a result, programs must now be actively parallelised before their performance can be increased to match the potential of these new multi-core processors.

Many have attempted to extract parallelism out of a sequential program [4, 9, 13, 12], but how much scope is there for further improvement? There have been several studies on the limits of parallelism in a sequential program, typically performed in one of two ways. One way is to re-schedule each dynamic instruction at the earliest cycle possible, using the execution trace of a previous run, and deriving the average number of instructions per cycle for the schedule [19, 10, 14]. The other is to construct *Dynamic Dependency Graphs* (DDGs) from the execution trace, from which the average "width" is calculated [2, 16]. The results from either method give a theoretical upper bound on the amount of average parallelism available in general programs, so that we have an idea of how much room there is for further improvement of parallelising compilers. Although both methods should give the same results, we have chosen the latter as it allows more flexibility to transform the graph in ways not possible with the former.

We evaluate the available average parallelism of benchmarks under several models, which differ by the types of dependencies considered and graph transformations applied. The results allow us to measure the effects different types of dependencies have on available parallelism. We find that when we consider only true dependencies—the *essence* of the underlying algorithm—the figures for average parallelism for many benchmarks are over 100.

In addition, we argue that certain parallelism-limiting dependencies are compiler-induced, and removing them would increase potential parallelism. For instance, certain name dependencies on a linear execution stack can be removed if a *spaghetti stack* is used instead. Thus one of our main contributions is to simulate the effects of using a spaghetti stack on available parallelism. This removes inter-frame name dependencies and breaks long true-dependency chains on the stack pointer. We find that for some benchmarks it results in a twofold increase in average parallelism. Taking this idea further, we find that if all address calculations are disregarded parallelism may be increased by up to an order of magnitude.


*WODA '09,* July 20, 2009, Chicago, Illinois, USA.

## 2. LIMITS OF PARALLELISM

### 2.1 Types of dependencies

Many forms of static analyses have been proposed for a compiler to extract parallelism out of a sequential program, with different degrees of speed-ups [4, 9, 13, 12]. It may be useful to have an idea of how much parallelism can potentially be extracted in this way, by evaluating limits of parallelism in general programs. In our analysis we do this by considering the parallelisation problem as one of minimising total execution time given the constraints imposed by its dependencies. Regions of code that are independent from each other can be executed in parallel. Parallelisation therefore involves exposing such independence and removing or minimising certain dependencies to allow more independent code to be exposed. We consider four types of dependencies—as illustrated in Figure 1.

- *True* (Read-after-Write) dependencies occur where one instruction uses a value produced by a previous instruction, and are generally considered to make up the essence of an algorithm. Most would consider true dependencies unremovable without changes on the algorithmic level. Later on, however, we show that such dependencies are sometimes introduced instead by the compiler and may be removed by modifying the compiler rather than the algorithm itself.
- *False* (anti/Write-after-Read) and *Output* (Write-after-write) dependencies, collectively known as *name* dependencies, occur due to the reuse of registers and memory locations. If two instructions which have such a dependency between them are executed out of order, use of incorrect values by an instruction may result. One way to remove such dependencies is to rename locations. For registers this means mapping the register in the instruction to a different register if there is no true dependency from an earlier instruction. In fact, register renaming is already performed in many modern processors (e.g. x86) and accordingly we would ignore false and output dependencies on registers in all of our analyses. The renaming of memory locations, however, is more difficult, as it may not be possible to work out the memory location accessed by an instruction until the instruction is executed.
- *Control* dependencies occur when whether an instruction executes depends on the result of a previous instruction, for example after a branch instruction. Control dependencies cannot be removed easily, because like true dependencies they capture some of the essence of a program (consider for instance the role of the control dependency in `y = x?1:0`), but their effects may be reduced by such techniques as speculative execution, where the result of an instruction is computed but not committed until we know it is safe to do so.

By considering different subsets of these dependencies, we could find out the effects each type of dependency has on available parallelism. The use of different models could be viewed as attempts to remove certain dependencies that we consider compiler artifacts, leaving only those that are part of the essence of the underlying algorithm of the original program. Ultimately we would like to explore the level of parallelism if all compiler-induced dependencies are removed.

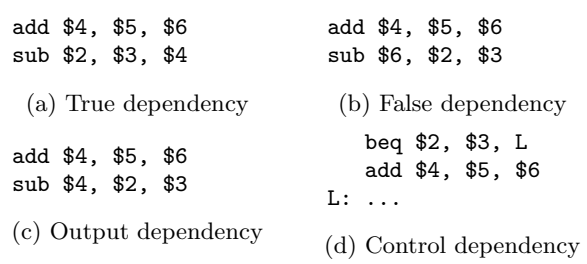


**Figure 1: Examples of the different types of dependencies. Examples here are written in MIPS assembly code, where the first operand is the destination register, and the rest are source registers.**

We do not view the results of this study as definitive limits—in particular we do not say that these limits are practically achievable (there may well be tighter architecture-dependent limits). For example, we do not consider any overheads related to threading, which one would imagine must exist and will impact speedup. Conversely these limits could also be exceeded by complex techniques, such as value prediction [11], which would invalidate our assumptions of dependency preservation. Our aim is only to evaluate limits of parallelism in existing programs in an architecture-independent way.

### 2.2 Dynamic Dependency Graphs

In a Dynamic Dependency Graph [2], a node represents the execution of an instruction, and edges represent the dependencies between instructions. Figure 2 shows an example of a simple DDG. We assume an idealised machine model where each instruction takes one cycle, but cannot be executed until all the instructions it depends on have been executed. We also assume an unlimited number of processors with zero inter-processor communication overheads. The minimum execution time therefore, is the number of instructions in the longest chain of dependencies, or in other words the critical path. Average parallelism, measured in instructions-per-cycle, is then calculated as the total number of instructions (graph size) divided by the minimum execution time (length of critical path).

Starting by considering all dependencies, we progressively remove certain types of dependencies, to evaluate the effects they have on parallelism. In addition to control and name dependencies, we also consider the effects of removing true dependencies on the stack pointer, by simulating the use of a *spaghetti stack*. This is because two procedure calls that are otherwise independent would be prevented from executing concurrently simply because the second call requires the value of the stack pointer, which would first be decremented and then incremented back by the first procedure call (see Figure 3). In the DDG this would show up as a true dependency from the second to the first procedure call. It can be argued however, that this dependency is only an artifact of the execution stack data structure introduced by the compiler, rather than part of the essence of the original algorithm used by the programmer. Indeed, if a linear stack is replaced by a spaghetti stack (also known as heap-allocated stack frames [1] or *cactus stack* [7]), such dependencies can be eliminated, along with name dependencies between different stack frames.

```
void main() {
  foo();
  bar();
}
```

(a) Code for two procedure calls inside `main` in C

```
 1 jal     foo          # main: call foo()
 2 addiu   $sp,$sp,-32  # foo: decrement stack pointer (new frame)
 3 addu    $fp,$0,$sp   # copy stack pointer to frame pointer
   ... <code for foo()>...
 4 addu    $sp,$0,$fp   # copy frame pointer to stack pointer
 5 addiu   $sp,$sp,32   # increment stack pointer (discard frame)
 6 jr      $ra          # return to main()
 7 jal     bar          # main: call bar()
 8 addiu   $sp,$sp,-32  # bar: decrement stack pointer (new frame)
 9 addu    $fp,$0,$sp   # copy stack pointer to frame pointer
   ... <code for bar()>...
10 addu    $sp,$0,$fp   # copy frame pointer to stack pointer
11 addiu   $sp,$sp,32   # increment stack pointer (discard frame)
12 jr      $ra          # return to main()
```

(b) Instruction trace of the execution of `main` in MIPS assembly

**Figure 3: An illustration of the true dependency chain on the stack pointer. The dependency chain in this case goes through lines 2, 3, 4, 5, 8, 9, 10 and 11.**

## 3. IMPLEMENTATION

We examined the execution of programs taken mostly from the miBench [8] suite. We chose this because while they provide a representative sample of real world applications, many of them also have data sets small enough for our analysis—currently our tool enables us only to examine program traces up to a few tens of millions of instructions long, and we prefer to analyse each program execution in its entirety. In addition we also analysed the synthetic benchmarks Whetstone [5] and Dhrystone [20].

The programs were compiled into MIPS binaries on Linux using gcc with uClibc as the standard library. The binaries were then simulated, mostly on QEMU [3], a uni-processor emulator capable of emulating most Linux system calls. The simulator would output a trace detailing the opcode of each instruction, as well as names of registers and addresses of memory locations that it has read from and written to, which is then analysed to build the DDGs.

| Model | True deps | Name deps | Control deps | Spaghetti stack | Ignoring address calcs |
|---|---|---|---|---|---|
| TruNamCtl | ✓ | ✓ | ✓ | | |
| TruCtl | ✓ | | ✓ | | |
| TruNam | ✓ | ✓ | | | |
| Tru | ✓ | | | | |
| TruNamSp | ✓ | ✓ | | ✓ | |
| TruSp | ✓ | | | ✓ | |
| TruNoAddr | ✓ | | | | ✓ |

**Table 1: Table comparing the seven models used in our analysis**

We built the graphs under several models which differed by the types of dependencies they considered, as well as the graph transformations applied, as detailed below, and summarised in Table 1. The critical path was then extracted for the graphs, enabling us to calculate the limit of parallelism as detailed above.

**TruNamCtl** This is the most restrictive (or pessimistic) model. We include in the graph all true dependencies for registers and memory locations. We also include name dependencies for memory locations but not registers, as it is relatively straightforward for register renaming to be performed in a modern processor (provided there are enough registers). Finally control dependencies are also included, which means that every instruction is dependent on the most recent branch or indirect jump instruction. This has the effect of limiting parallelism to within a dynamic basic block[1].

**TruCtl** Name dependencies for memory locations are ignored, simulating the effects of perfect memory renaming.

**TruNam** Control dependencies are excluded instead (e.g. by perfect branch and jump prediction), allowing parallelism beyond basic blocks to be exhibited.

**Tru** Only true value dependencies are considered here, the "essence" of the program.

**TruNamSp** Here we model what happens when a spaghetti stack is deployed, in order to reduce compiler-induced dependencies on the stack pointer. Two transformations are applied to the graph. The first transformation removes all inter-frame name dependencies on the stack, as these different frames which used to occupy the same area on the stack would now occupy different areas in the heap. In the second transformation, every instruction that decrements[2] the stack pointer has been replaced with a `malloc_frame` pseudo-instruction. Unlike the original instruction, the new `malloc_frame` instruction has no true dependency on the previous value of the stack pointer, but simply returns the address of a free area in the heap that can be used as a new stack frame, thus breaking true dependency chains

[1]A dynamic basic block is defined as the sequence of instructions between two branch/indirect jump instructions. As opposed to static basic blocks, the target of a branch not taken does not begin a new basic block.

[2]In a MIPS architecture the execution stack grows downwards so a push on the stack would decrement the stack pointer.

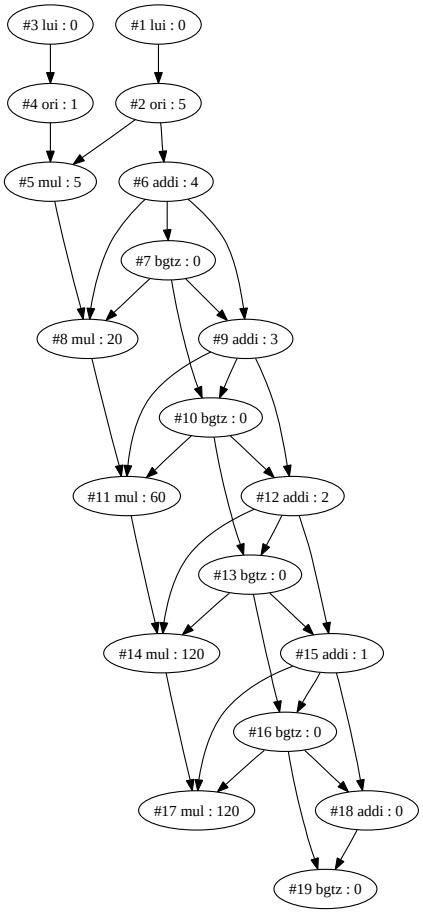


**Figure 2: An example of a DDG for the calculation of the factorial of 5. Edges out of the `bgtz` instructions are control dependencies—all other ones are true dependencies. Nodes with no outgoing edges (the `mul` and `bgtz` nodes in the bottom) are considered results of the calculation.**

on the stack pointer. We ignore the overhead this instruction might cause in practice, and assume this instruction takes one cycle, just like all other instructions.

**TruSp** In this model we look at the effect of a spaghetti stack on the DDG when only true dependencies are considered.

**TruNoAddr** Taking the idea of removing compiler artifacts one step further, we consider the effects of removing address calculations from the graph altogether, claiming that the stack pointer is just one example of address calculations that are only compiler artifacts rather than an essential part of the algorithm.[3] This would show the amount of parallelism available in a world where memory address values do not have to be calculated at runtime, but can rather be predetermined statically (as in a language with non-recursive static stack allocation such as FORTRAN). It can be argued that such a measure is unrealistic, given that *some* address calculations are in fact an integral part of an algorithm, e.g. radix sort. Nonetheless, it still provides an estimate of the upper bound of parallelism (albeit not the tightest one) if compiler artifacts are removed or replaced by data structures (such as the spaghetti stack) which are less parallelism-limiting. Note that although this model subsumes the TruSp model, removing nodes has the effect of reducing the graph size, so if the critical path is not affected significantly by this transformation, the average parallelism could in fact decrease slightly compared to TruSp, even if the program runs faster overall. If address calculations made up a large part of the critical path, however, parallelism would still rise.

[3]Another example is the heap memory allocator, which would update some internal data structure (such as a linked list) every time it is called, giving rise to true dependencies between calls, even though commuting the orders of these calls would make no difference to the semantics of the program (assuming sufficient memory is available).

# 4. RESULTS

Figure 4 shows the results of our analysis under the seven models, from which we focus on specific areas to draw several observations.

## 4.1 Effects of control dependencies

By comparing models TruNamCtl and TruCtl with TruNam and Tru respectively in Figure 4, we see the effects of control dependencies on available parallelism. It shows that parallelism is severely restricted in the presence of control dependencies, as they prevent code from different basic blocks being executed in parallel, leaving only intra-(dynamic) basic block parallelism. With control dependencies (models TruNamCtl and TruCtl) most benchmarks only exhibit average parallelism (instructions per cycle) of less than ten, on which the removal of false and output dependencies have no effect. This generally confirms Wall's findings [19], although exact comparisons are not possible. In fact a lot of this *intra*-basic block parallelism is already realised in multiple-issue processors. In order to extract more significant amounts of implicit parallelism, therefore, we need to look far beyond basic block boundaries for parallelism at a coarser granularity.

Despite this low limit, fortunately there are already techniques which allow us to safely violate some of these control dependencies, resulting in parallelism higher than what is suggested here. The first of these is the successful use of branch prediction in computer hardware, which results in high rates of correct prediction (at least 80% and often over 95%) with relatively straightforward schemes [15]. This allows instructions following a branch instruction to be scheduled before the branch instruction itself, and discarding the results of such speculative instructions if the prediction turns out to be wrong. Nevertheless, even perfect branch prediction would not be able to eliminate all control dependencies, as it tends to apply only locally—in other words, the window size[4] is still limited. This means that instructions that are more than a few branches away cannot be scheduled before all these branches even in the absence of other dependencies.

Secondly, it can also be argued that some *inter*-basic block

[4]In hardware scheduling, the window size is the maximum number of pending instructions that are considered for scheduling at any one time.

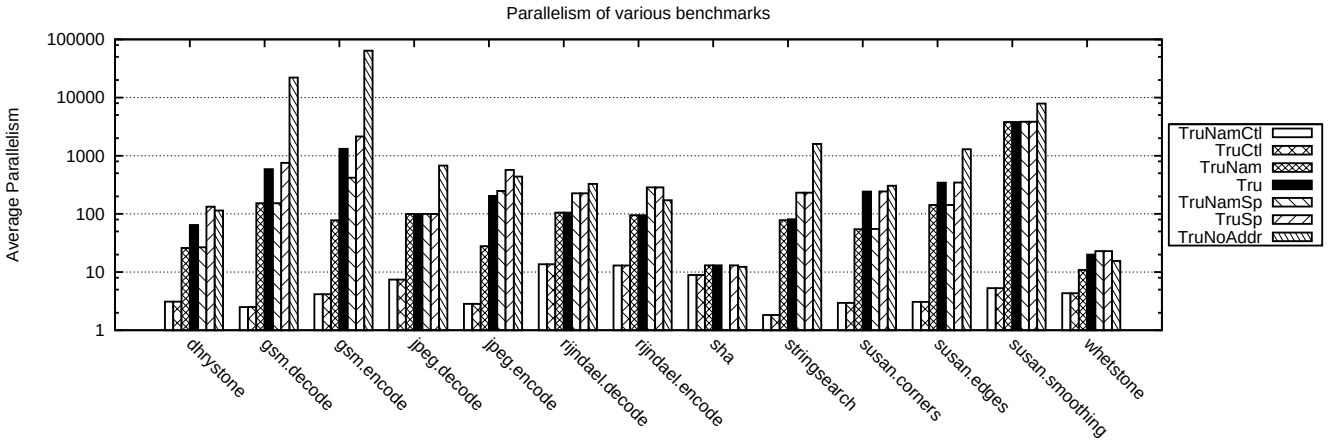


**Figure 4: Parallelism of various benchmarks for the seven models**

parallelism can be extracted statically. In particular, with static control dependence analysis we can mark control dependencies more precisely by eliminating control dependencies after merge points. To illustrate this, in a program of the form `if R1 then { R2 } else { R3 } R4` we would be able to remove the control dependency from `R4` to `R1` because control flow merges at that point[5]. This information can be extracted using a Program Dependence Graph [6], and it will be interesting to extend our analysis to examine the increase in parallelism resulting from its use. However, past research suggests the impact of such static analysis is limited in this respect, especially regarding general non-numeric applications [10, 17]. This would mean that dynamic analysis must play a role in reducing the effects of control dependencies, for instance through thread-level speculation.

## 4.2 True dependencies only

As mentioned earlier, while control dependencies can be speculated away, and false and output dependencies can be reduced with register and memory renaming and other compiler optimisations, true dependencies represent the true essence of a program and generally cannot be removed. The Tru model should therefore give us a picture of the limits imposed on parallelism by the algorithm only. Figure 4 shows that under this model, many programs exhibit average parallelism of over 100. This is much greater than speed-ups reported for existing implementations of parallelising compilers, which tend mostly to be in single figures [4, 9, 13, 12] showing that there is still a big gap between the amount of speed-up we can achieve with current parallelisation techniques and what is theoretically possible. This gap means that there is still much potential parallelism to be exploited, even if such high parallelism may not be achievable without drastic changes to compilation methods or even models of computation.

## 4.3 Removing compiler-induced dependencies

We argue that even some of the true dependencies are only compiler artifacts and do not constitute the essence of the underlying algorithm as specified by the programmer. One example of this is the use of the execution stack and the stack and frame pointers. By comparing models TruNam and Tru with TruNamSp and TruSp respectively, we see how the limits of parallelism change when a *spaghetti stack* is used instead of a linear one. This has effects on two types of dependencies: Firstly name (false and output) dependencies on the stack across different frames are removed as these frames would now reside on different areas of the heap. Secondly true dependencies on stack pointer decrements are also removed as they are replaced with `malloc_frame` pseudo-instruction nodes. Figure 5 shows graphically the effect this replacement has on a synthesised program with two identical procedure calls.

Our results in Figure 4 show a mixed picture with this optimisation. For some benchmarks (jpeg.decode, sha, susan), the spaghetti stack makes no difference at all to the available parallelism. For some others (dhrystone, gsm.decode), parallelism only increases with the spaghetti stack when we consider only true dependencies—name dependencies on the heap and within the same frame would otherwise remain as the bottleneck, preventing the spaghetti stack from affecting the overall parallelism. But for the remaining (rijndael, gsm.encode, jpeg.encode, stringsearch, whetstone) we have a twofold increase in parallelism even if we still consider false and output dependencies on the heap. This is good news as it shows that even under more realistic assumptions a relatively straightforward change to the compiler can still result in a doubling of available parallelism. It needs to be noted however that this model was used under the assumption that each `malloc_frame` instruction, like all other instructions modelled, takes one cycle to execute, and does not cause other dependencies. In practice it is likely that this instruction will take much longer, and depending on its implementation may necessitate new instructions not modelled in our graphs.

By comparing models Tru and TruNoAddr in Figure 4 we see the effects of ignoring address calculations by removing from the graph all instructions the results of which are subsequently used anywhere as addresses for memory accesses. In some cases (most notably gsm and stringsearch) this has resulted in an increase of an order of magnitude over the Tru model. In other cases there is in fact a slight drop in parallelism, showing that the drop in the critical path length was not great compared to the drop in overall graph size.

[5]Our dynamic-only analysis cannot identify path merge points, and would therefore require static analysis to eliminate such a dependency.

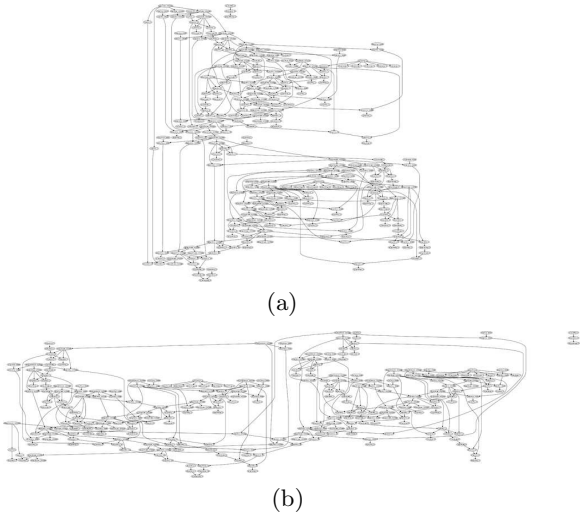

**Figure 5: Shape of Dynamic Dependency Graph of a program with two identical procedure calls (a) without and (b) with the "spaghetti stack" transformation. Note the relative width (average parallelism) and depth (length of critical path) of the two graphs.**

Generally, however, we see high average parallelism rising above 1000 in a few cases.

## 5. RELATED WORK

Wall [19] produced a comprehensive study on the limits of instruction-level parallelism in SPEC benchmarks. Using an oracle based on the execution trace of a benchmark, he used a greedy algorithm to schedule each instruction at the earliest possible cycle. He examined the effects on available parallelism of popular optimisations, such as alias analysis, register renaming and branch prediction, as well as practical limits on window size (the maximum number of instructions considered for scheduling at any one time) and cycle width (the maximum number of instructions that can be scheduled to run in the same cycle). He found that branch and jump prediction resulted in the highest gain in parallelism in programs, followed by register renaming and alias analysis. He also discovered that parallelism was significantly limited when window size was restricted, meaning that much of the parallelism related to instructions that were very far apart. While Wall's concluding view on available parallelism was ambivalent, we believe that by shifting the focus from instruction-level parallelism in superscalar processors to coarser-grained parallelism in multi-core processors new possibilities can be explored which had been thought difficult to achieve at the time. Our work extends the scope of Wall's study, for instance by also renaming memory locations and considering compiler artifacts.

Lam and Wilson [10] examined in more detail the effects of control dependencies on parallelism. Using a similar method to Wall's they looked at parallelism when using a mixture of control dependence analysis, multiple flows of control and speculative execution. False and output dependencies were ignored, as were true dependencies on the stack pointer and those on call/return instructions. Their conclusions underlined the importance of speculative execution to realising sufficient parallelism.

Austin and Sohi [2] was the first to study limits of parallelism using Dynamic Dependency Graphs (DDGs) of assembly instructions. Their study largely assumed control dependencies were perfectly resolved, and focused on the effects of false and output dependencies on available parallelism. They found that register renaming alone resulted in good levels of parallelism, but more is exposed when memory locations are also renamed. They also obtained a profile of the parallelism of the program over its runtime, and found that parallelism is often unevenly distributed and bursty. Our Dynamic Dependency Graphs are similar to the ones used there, but we have extended their analysis by also examining control dependencies and address-related true dependencies.

Postiff et al. [14] ran a similar analysis using Wall's method and ignoring control dependencies. They reached similar conclusions to Austin and Sohi regarding register and memory renaming. However, they also looked at true dependencies on the stack pointer, and found significant gains in parallelism in some cases when they are eliminated. Our methodology of using DDGs is different, but our results generally agree with theirs. A significant difference between our approach and theirs is that while Postiff et al. discarded all true dependencies on the stack pointer, we only discard *decrements* (that is, frame allocations) but not deallocations. This is because under the *spaghetti stack* model that we use, the stack pointer still needs to be restored to point to the calling frame when a procedure returns—this is a true dependency that cannot be resolved simply by using the *spaghetti stack*. Removing dependencies on frame deallocations would require further changes to the compiler, but may result in additional parallelism. Our work also extends this one by considering control dependencies and examining the effects of ignoring all address-related true dependencies (not just on the stack pointer).

Stefanović and Martonosi [16] also created DDGs to look at the effects of eliminating address calculations. This was done by removing loads and stores from the graph, replacing them with an edge that goes directly from the producer instruction of the value to the consumer instruction. Instructions from which values are used only as addresses to loads and stores are also removed. They reported vast improvements in parallelism in some cases (but slight deterioration in others where address calculations do not form long dependency chains). Our TruNoAddr model is similar to this analysis, with the exception that only address calculation instructions are removed and not loads and stores, as we reason that loads and stores are still required even if we can now statically determine addresses rather than calculating them at runtime.

We claim that our research is a unified framework that explores ideas in the models presented in these previous studies, allowing like-for-like comparison between such models.

## 6. CONCLUSIONS AND FUTURE WORK

We have examined the limits of parallelism in benchmark programs, and the effects of various types of dependencies on them, by constructing Dynamic Dependency Graphs from execution traces. As with previous studies that we extend, we find that control dependencies form the biggest obstacle to realising more than a modest amount of parallelism

(above single figures). By disregarding control dependencies the available parallelism is much greater, even more so when memory locations are renamed to remove false and output dependencies. In fact, we find that parallelism for many programs reaches above 100 when only true dependencies are considered. We have also looked at true dependencies on the stack pointer, reasoning that these are not an essential part of an algorithm but are only compiler-induced. By using a *spaghetti stack* to remove some of these dependencies, we have managed to double available parallelism in some programs. It would be interesting to scale up our experimental framework, in order to extend our analysis from the embedded benchmarks of miBench into server benchmarks. Another possible direction is to try to separate the parallelism found here into different granularities, as an indicator of how the parallelism is best exploited, e.g. whether with superscalar processors or multi-cores. We view our findings with qualified optimism, in that while the limits of parallelism are generally quite high, achieving it in practice is not straightforward, and will require significant changes to the current method of compilation or even model of computation. Nevertheless, we find that in most of these benchmark programs much parallelism is there—the challenge is to find better ways to exploit it.

## 7. ACKNOWLEDGEMENTS

The first author would like to thank Christian Fensch, David Greaves, Daniel Greenfield, Anton Lokhmotov, Simon Moore, Robert Mullins, Matthew Parkinson and David Ung for their help and advice, and for St John's College, Cambridge for its financial support.

## 8. REFERENCES

[1] A. W. Appel and Z. Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS–TR–450–94, 1994.

[2] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Nineteenth International Symposium on Computer Architecture*, pages 342–351, Gold Coast, Australia, 1992. ACM and IEEE Computer Society.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[4] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoe, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[5] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *Computer Journal*, 19(1):43–49, 1976.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[7] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[9] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[10] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Nineteenth International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, 1992. ACM and IEEE Computer Society.

[11] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. *SIGOPS Oper. Syst. Rev.*, 30(5):138–147, 1996.

[12] G. Ottoni and D. I. August. Global multi-threaded instruction scheduling. In *Proceedings of the 40th annual IEEE/ACM International Symposium on Microarchitecture*, pages 56–68, 2007.

[13] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[14] M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge. The limits of instruction level parallelism in SPEC95 applications. *Computer Architecture News*, 217(1):31–34, 1999.

[15] J. E. Smith. A study of branch prediction strategies. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, New York, NY, USA, 1998. ACM.

[16] D. Stefanović and M. Martonosi. Limits and graph structure of available instruction-level parallelism (research note). *Lecture Notes in Computer Science*, 1900:1018–1022, 2001.

[17] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *SIGARCH Comput. Archit. News*, 28(2):1–12, 2000.

[18] H. Sutter. A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):16–20, March 2005.

[19] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 176–189, New York, NY, 1991. ACM Press.

[20] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.