Chip Multi-Processor Scalability for Single-Threaded Applications

Neil Vachharajani*, Matthew Iyer†, Chinmay Ashok†
Manish Vachharajani†, David I. August*, and Daniel Connors†

*Department of Computer Science
Princeton University
{nvachhar,august}@princeton.edu

†Department of Electrical and Computer Engineering
University of Colorado at Boulder
{iyer, ashokc, manishv, dconnors}@colorado.edu

Abstract

The exponential increase in uniprocessor performance has begun to slow. Designers have been unable to scale performance while managing thermal, power, and electrical effects. Furthermore, design complexity limits the size of monolithic processors that can be designed while keeping costs reasonable. Industry has responded by moving toward chip multi-processor architectures (CMP). These architectures are composed from replicated processors utilizing the die area afforded by newer design processes. While this approach mitigates the issues with design complexity, power, and electrical effects, it does nothing to directly improve the performance of contemporary or future single-threaded applications.

This paper examines the scalability potential for exploiting the parallelism in single-threaded applications on these CMP platforms. The paper explores the total available parallelism in unmodified *sequential applications and then examines the viability of exploiting this parallelism on CMP machines. Using the results from this analysis, the paper forecasts that CMPs, using the "intrinsic" parallelism in a program, can sustain the performance improvement users have come to expect from new processors for* only *6-8 years provided many successful parallelization efforts emerge. Given this outlook, the paper advocates exploring methodologies which achieve parallelism beyond this "intrinsic" limit of programs.*

I. Introduction

Historically, semiconductor process and technology improvements have provided microarchitects with smaller, faster, and less power-hungry transistors. The responsibility of microarchitects was to translate these device-level improvements into cheaper and faster microprocessors. As device technology scaled, microarchitects could design processors using many more transistors without dramatically increasing the cost of the product. Consequently, designers used this glut of newly available fast transistors to build high-clock speed, deeply pipelined, wide-issue, out-of-order, speculative microprocessors. Through these improvements in both physical design and microarchitecture, processors in use today are almost 100 times faster than the processors of just 15 years ago [24].

Unfortunately, this singular focus on processor performance is no longer tractable. Today designer's must combat overwhelming design complexity, unmanageable power requirements, thermal hot-spots, current-swings, signal noise and other effects when designing a faster processor. Consequently, the techniques previously employed by designers are showing diminishing returns. For example, several Intel designers observe that the design complexity of a processor's instruction window and the delay through the window scales quadratically with the number of instructions it can hold while delivering less than linear performance improvement [32]. In response to these design challenges, many microprocessor designers and manufacturers have turned to building chip multi-processors [1], [13], [16], [36]. These products reduce design complexity by replicating a design unit (the processor core) across the die. The designs also help allevieate thermal issues by spreading computational resources more uniformly. And, if all cores can be kept occupied, the performance per unit power, is kept almost constant. Consequently, these designs create far fewer complications for microprocessor manufacturers. The key to the success of these designs, however, lies in identifying workloads that can leverage the additional processor cores, or *transforming* existing workloads into ones that can.

Multi-programmed systems and throughput driven applications *naturally* fit the programming model offered by CMPs and consequently, their performance will benefit from these new architectures. Unfortunately, the latency of any single task remains unchanged. At best, the common single-threaded applications will see no benefit from these future architectures. At worst, they will experience heavy slowdowns as the aggressiveness of each core is scaled back to make room for larger numbers of cores. To provide the consistent speedup of single-threaded applications that users have come to expect (i.e. to continue the Moore's law of single-threaded application performance), techniques to reduce the latency of single-threaded applications using CMP architectures must be developed.

Decades of research in automatic parallelization of single-threaded applications has yielded some limited success. Scientific applications, in particular, or more generally, programs with regular accesses to memory and regular control flow have been successfully parallelized [3]. General-purpose programs, which contain indeterminate memory accesses and control flow, however, have seen far less success [26]. Researchers have turned to thread-level speculation (TLS) [11], [15], [22], [26], [27], [35] and speculative pre-computation [17], [21], [34], [39], [40], [41] to leverage CMP architectures for accelerating general-purpose programs. Each technique attempts to use additional processor cores to either run large blocks of code speculatively or to run code to enhance the performance of the main processing thread (e.g. warm the caches or branch predictor).

Rather developing mechanisms to extract parallelism from general-purpose, single-threaded applications, other researchers have attempted to quantify the "intrinsic" amount of parallelism found in these applications. Instruction-level parallelism (ILP) limit studies [2], [5], [14], [18], [19], [25], [30], [37], [38] show that in ideal circumstances, these same hard to parallelize general purpose programs can execute, on average, as many as a few hundred to a few thousand instructions each cycle. Unfortunately, existing ILP architectures have been unable to unlock this parallelism because they process instructions along a single instruction stream. Consequently, instructions that are not ready to execute occupy valuable resources that prevent other ready, independent instructions later in the stream from entering the pipeline. The growing disparity between processor speed and memory speed exacerbates this resource congestion since load operations that miss in the cache and all subsequent instructions remain in the pipeline for significantly more time.

Techniques such as kilo-instruction processors [7], [9] attempt to overcome this in-order instruction processing but unfortunately these solutions do not address the other challenges (heat dissipation, power consumption, design complexity, etc.) facing aggressive single core designs. On the other hand, multi-processors in general, and CMPs, in particular, naturally avoid in-order processing of instructions by executing multiple streams of instructions concurrently. Consequently, if the parallelism observed in ILP limit studies could be expressed to a CMP as threads, then a significant improvement in single threaded application performance may be observed when these applications are parallelized and run on a CMP.

Despite this potential, it remains an open question to determine how much of this ideal parallelism can be extracted by CMP architectures. Moreover, it is unclear how many processor cores are necessary to extract this performance. Answers to these questions will allow designers and architects to understand how many process generations worth of transistors can be dedicated to additional processor cores while still improving single-threaded application performance. Thus, answers to these questions provide a timeline that indicates when architects and programmers can no longer rely on performance improvements from exposing the intrinsic parallelism in *unmodified* programs to CMP machines.

This paper attempts to approximate these values thus giving computer architecture researchers advance warning about how soon new techniques must emerge from our community. To establish an idealistic limit on the amount of parallelism present in unmodified, general-purpose, single-threaded applications, Section II presents a limit study reminiscent of those found in the literature. Section III then measures how much of this parallelism can be exploited by various idealized CMP architectures using greedy scheduling of dynamic instructions under various cost models. Section IV discusses the implications of the results presented in the prior sections and outlines possible future strategies that may help to continue performance scaling of single threaded applications. Finally, Section V summarizes the results of the paper and restates the position of the authors.

II. The Availability of Parallelism

Figure 1 shows the available ILP for several benchmarks from the SPEC CINT2000 suite using the train input sets. To compute the instructions-per-cycle (IPC), complete program traces were collected for each program execution and these traces were ideally scheduled respecting differing dependences and resource constraints ranging from extremely optimistic to dependences and constraints reminiscent of today's processors. In all cases, each instruction was assumed to have unit latency and all true data dependences were respected. A varying re-order buffer (ROB) size constraint was imposed to illustrate the effects of myopic processors. This constraint limited how far into the trace the scheduler could look, and, as a side-effect, limited the maximum number of instructions that could be scheduled during any cycle. Instructions were placed in *and* removed from the simulated ROB in trace order. Schedules were created for ROB sizes of 128, 256, 512, 1024, 10240, and infinite. For each ROB size, schedules were created respecting *true* control dependences (denoted C in the figure), predicting control dependences and respecting only

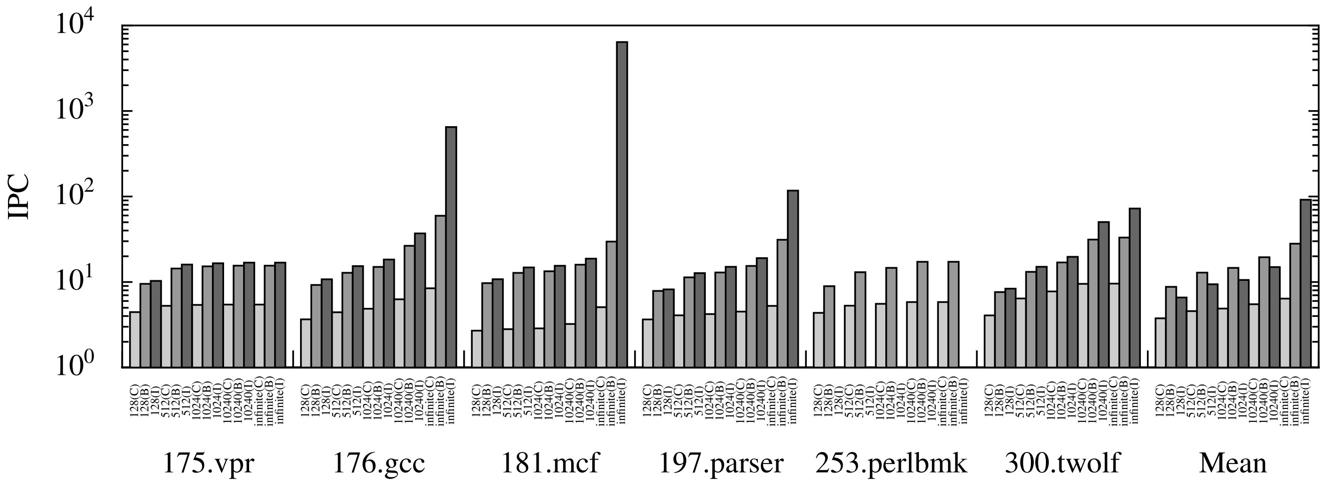

Fig. 1. Non-Speculative Parallelism in Unmodified SPEC Programs

those that were mispredicted (B), and ignoring *all* control dependences (I).

Similar to limit studies of the past [2], [5], [14], [18], [19], [25], [30], [37], [38], the figure clearly illustrates that in most cases significant parallelism exists (note the log-scale y-axis) provided that a sufficiently large window of instructions can be considered for scheduling. If control dependences are ignored, programs such as 176.gcc, 181.mcf, and 197.parser achieve IPCs in excess of one hundred and in some cases in excess of one thousand. As previously observed [30], when the effects of control dependences (even when mitigated by branch prediction) are considered, window sizes of about ten thousand instructions are able to capture the vast majority of parallelism present in these programs. Only the applications 176.gcc, 181.mcf, and 197.parser possess parallelism beyond 10240 instructions (61%, 58%, and 63% more than achived by the 10240 instruction ROB respectively), but even in these cases, significant parallelism is observed with ROB sizes of 10240.

Unfortunately, the ILP measured when ignoring control dependences seems unattainable. For example, in 181.mcf, if all control dependences are ignored except for those due to branches in the *sort_basket* function (9 static branches consisting of 13.34% of the dynamic branches in the program), the IPC drops from 6397.25 to 25.41. These branches represent control decisions in a quick sort algorithm making them difficult to predict. It is unlikely that future advances in branch prediction will be able to overcome this type of information-theoretic limitation in predictability.

The limitations due to ROB size, however, may be surmountable. While it is unlikely that processors will be built with ROBs that hold thousands or tens of thousands of in-flight instructions, chip multi-processors combined with application threading strategies may be able to extract the parallelism revealed by this study using many small (i.e. 128 instruction) windows. The next section quantifies this promise by measuring the parallelism achieved when dynamic instructions are scheduled across many independent ROBs using simple greedy algorithms.

III. Parallelism in CMPs

The previous section illustrated the presence of parallelism in conventional, hard-to-parallelize general purpose programs. This section quantifies the promise of CMPs to realize this parallelism. To that end, we present a simple greedy CMP instruction scheduling algorithm, evaluate the parallelism it realizes, and finally measure the effect on parallelism of several constraints that may hinder performance. Throughout this section, we will analyze the scalability of the CMP solution and identify "knees" in the performance curves; that is to say, we identify the points at which additional processor cores provide little to no performance advantage.

A. CMP Instruction Scheduling

The principle advantage CMPs offer over single-core processors is the ability to avoid resource congestion created by contiguous instructions that form long dependence chains. These chains prevent the processor from seeing independent instructions that lie farther in the instruction stream. Multi-core architectures, on the other hand, avoid this bottleneck since multiple independent, non-contiguous windows of instructions can be analyzed, one window per core. Since instructions in different cores retire independently, bottlenecks in one core do not affect other

Algorithm 1 CMP Instruction Scheduling Algorithm

```
for $instr \in trace$ do
  $best\_ROB \leftarrow 0$
  $min\_time \leftarrow \infty$
  for $i = 1$ to $n$ do
    $time_i \leftarrow \mathsf{try\_ROB}(instr, ROB_i)$
    if $time_i < min\_time$ then
      $min\_time \leftarrow time_i$
      $best\_ROB \leftarrow i$
    end if
  end for
  $\mathsf{schedule\_insert}(instr, ROB_{best\_ROB}, time_{best\_ROB})$
end for
```

cores. To evaluate the potential parallelism CMPs offer to execute single-threaded applications, the instruction scheduler used for the limit study in the previous section must be augmented to be aware of multiple ROBs.

Algorithm 1 presents the algorithm, a variant of the ETF (Earliest Time First) algorithm [12], used for scheduling. The scheduler iterates over the instructions in the program trace. For each instruction, the try_ROB function computes the cycle in which the instruction would be scheduled if it were allocated to i^{th} core. As each core is tried, the core which executes the instruction earliest is recorded. After all cores are tried, the instruction is scheduled into the earliest possible slot, and the occupancy for the ROB for the corresponding core is updated by the schedule_insert function.

Intuitively, the algorithm will allocate instructions to a single core until that core (i.e. the core's ROB) gets backlogged with a long dependence chain. In such a case, the scheduler opts to allocate the instruction to another core which has the least resource contention. In this way, the scheduler greedily attempts to get each instruction to execute as early as possible. This greediness may result in suboptimal schedules, however, as the results will indicate, the scheduler performs quite well. The reader should note that more intelligent scheduling algorithms may improve the parallelism observed, but such improvements further support the claims made in the paper.

Notice that the net effect of this scheduling process is reminiscent of TLS architectures. The sequential semantics of the original single threaded program are preserved; the instructions are merely scheduled to different cores and inter-core (register and memory) dependences are satisfied through inter-core communication. Unlike TLS, however, the scheduler does not need to speculate dependences. Since it possesses oracle knowledge, it can allocate the instruction to any core, balancing the time when an instruction's dependences are satisfied with the time it will enter the ROB of the core to which it is allocated. Additionally, the scheduling algorithm is not constrained to allocate instructions to cores in a round-robin fashion as is typically required in TLS architectures. These limitations of TLS, as well as misspeculation penalties, may make it difficult to realize the performance shown here with TLS architectures, but the likeness to these architectures may help make the results more concrete in the reader's mind.

B. Extractable Parallelism

Figure 2 shows how the SPEC CINT2000 benchmarks evaluated earlier fare on the CMP model. The experiments in this section and for the remainder of the paper respect control dependences, perform branch prediction, and avoid stack related dependences [18], [25]. In particular the numbers presented in this graph respect only *true* control dependences. That is to say, control independent instructions following a branch are not squashed if the branch is mispredicted. Further, this data assumes that the inter-core and intra-core inter-instruction latencies are identical (i.e. there is no additional inter-core latency). The first four bars (from left to right) and the last bar for each benchmark, show the performance of the application for a single ROB of size 128, 256, 512, 1024, and infinite instructions respectively. The remaining bars in the graph show the parallelism obtained in various CMP configurations. Each configuration is denoted nxs where n is the number of cores and s is the size of the ROB for *each* core. The graph presents results for 2x64, 2x128, 4x64, 4x128, 8x32, 8x64, 16x32, and 16x64 configurations. Notice that the CMPs show significant potential. In every case, a CMP with the same number of total instruction window entries as the single window machine always outperforms the single window. For example, in 175.vpr the 2x64 CMP (128 total instruction slots) outperforms the 128 entry ROB model. Even more striking is that the 4x64 CMP outperforms the 1024 instruction monolithic window in all cases. Often times, the 16x64 configuration performs almost as well as the infinite window!

The graph also illustrates that in many cases, only a few cores are needed to extract most of the available parallelism. For 175.vpr, the 2x128 CMP provides nearly all of the parallelism present in the application. For 3 of the 5 remaining benchmarks (181.mcf, 197.parser, and 253.perlbmk), the 8x64 configuration captures nearly 90% of the parallelism provided by the 16x64 configuration and much of the total available parallelism. The remaining two benchmarks (176.gcc and 300.twolf) benefit significantly in the 16x64 configuration achieving 75% and 96% of the available parallelism respectively. This data clearly suggests that additional cores rapidly provide diminishing returns, and if these applications are representative, that a 16 core CMP should be adequate for extracting the vast majority of parallelism from single-threaded, general-purpose applications.

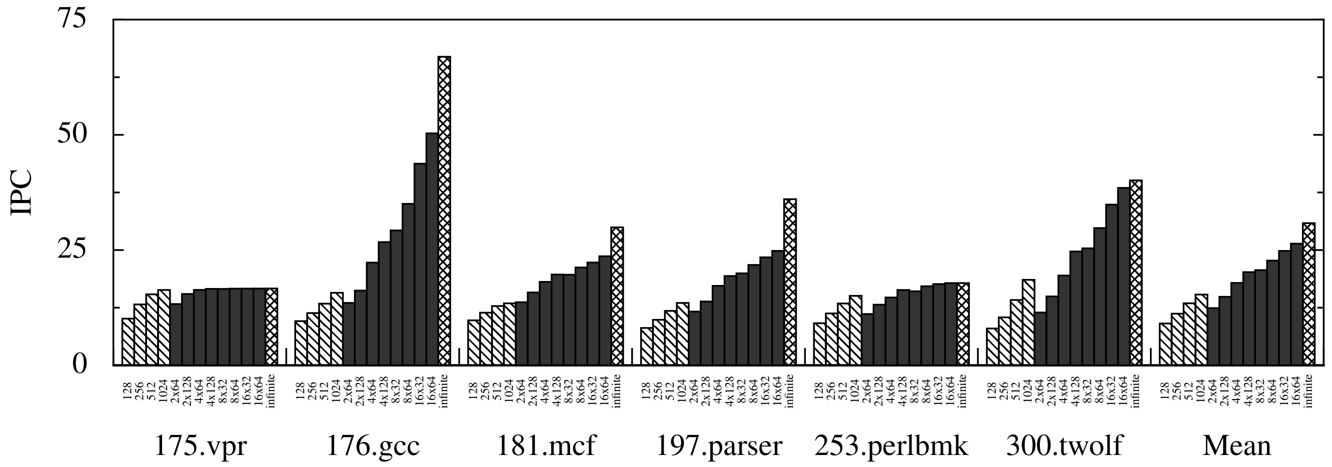

Fig. 2. Comparison of Parallelism Exploitable by Different Window Sizes

C. Inter-Core Communication Latency

While the above performance may be observed on an actual CMP, the results are likely exaggerated since inter-core communication incurred no penalty. In practice, the inter-core communication delay will reduce the observed benefits, possibly severely. In this section, we revisit the performance of CMPs when inter-core communication is not free.

To account for communication latency, Algorithm 1 must be enhanced to understand when dependences are being satisfied from different processor cores. In the algorithm, the try_ROB function computed the execution time of the given instruction when scheduled to a particular ROB. This function returned the maximum of the time an instruction's dependences were satisfied and the time the instruction could enter the given ROB. To account for communication latency, the time an instruction's dependences are satisfied must be delayed if a particular dependence is satisfied from another core.

We model a single fixed communication latency between all cores (with no bus or network contention) and add this latency to the time a particular operand is ready if the operand is being obtained from a different core from the one where the instruction is being executed. Just as before, the scheduler will try all cores and choose the one that allows the instruction to run the earliest. Intuitively, this will create an affinity for instructions to remain on the core where their data sources executed. Only if that core experiences a back-log of instructions larger than the communication latency will an instruction move to a new core.

Figure 3 shows the same benchmarks from the previous section executing with the communication latency model just described. The single ROB IPC bars for each benchmark are the same as in Figure 2 and are reproduced for reference. For the ROB configurations 2x64, 4x64, 8x64, and 16x64, the IPC is shown for three different latencies: 16, 4, and 0. The 0-latency case is identical to the CMP bar from Figure 2 and is also reproduced for reference.

The graph illustrates that communication latency has significant deleterious effects for the CMP models. In particular, with a 16 cycle latency, a CMP now only performs on par with or slightly outperforms the single-window machine with an equal total number of ROB entries. With latency 4, to outperform the 1024 entry single-ROB, often times 8 cores with a ROB size 64 are necessary.

Despite the obvious deleterious effects, one should note that the trend of diminishing returns remains. While a communication latency of 16 reduces the overall parallelism observed by the CMP, it is unlikely that 32 or 64 cores will perform significantly better than 16. Consequently, even under these conditions, an 8-core CMP seems to be a sweet-spot for performance and 16-cores seems to promise near optimal results given the particular partitioning algorithm.

D. Control Speculation and Recovery Techniques

Finally, this section examines how control dependences are handled in the model and contrasts this with more practical misspeculation recovery techniques. In the current models, only true control dependences that are misspeculated are respected. While this mimics modern machines allowing control dependent instructions to be executed early provided the dependence is predicted correctly, it behaves optimistically with respect to mispredicted branches. In particular, control-independent regions of code following a hammock (if-then-else statement) are free to be

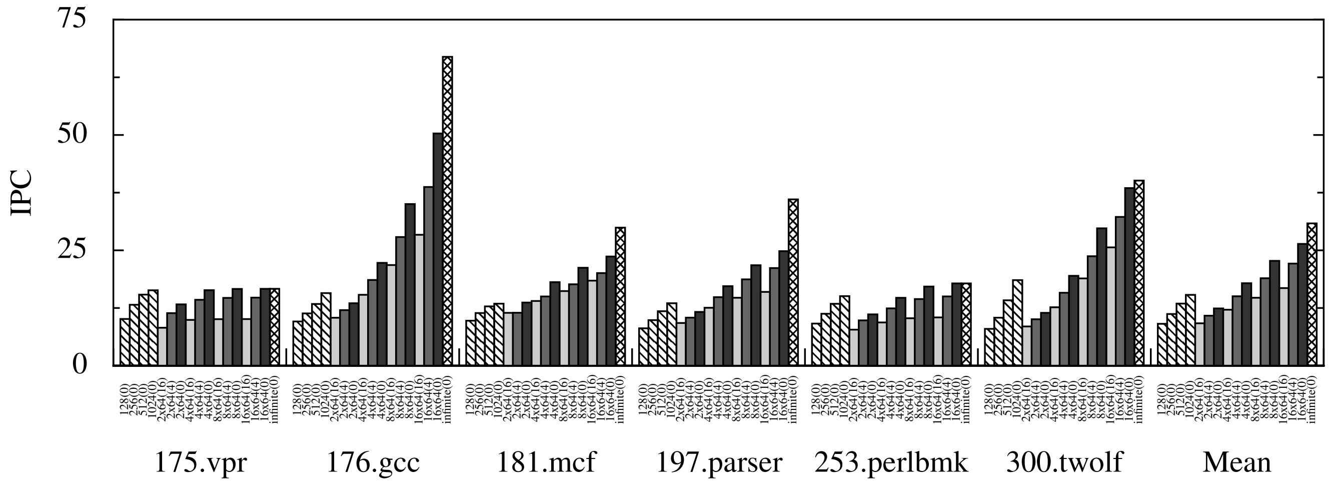

Fig. 3. Comparison of Parallelism Exploitable by Different Latencies

hoisted above branches, *even* if the branch is mispredicted. While such selective replay recovery mechanisms have been proposed [6], [10], most aggressive processors simply squash *all* instructions following a misspeculated branch due to the complexity of identifying control re-convergence and re-inserting instructions into the *middle* of the scheduling window.

To model the more conservative recovery mechanism, the dependences used by the scheduler must be augmented to include a dependence arc between a mispredicted branch and all subsequent instructions. To model core-to-core independence, these arcs are only drawn between subsequent instructions scheduled in the *same* core as the branch. Subsequent, control-independent instructions in other cores remain independent of the branch. The scheduler will consider these dependences and the constraints they impose in the same way as before; an instruction may be scheduled to a particular core because it is unhindered by a particular branch misprediction. Note that this style of recovery is again reminiscent of TLS architectures which often do not squash later threads in response to an intra-thread branch misprediction of an earlier thread.

Figure 4 illustrates the affect of this branch misspeculation recovery policy. For each window size, the graph contains two bars, one for the previous selective-replay-like branch recovery methodology (denoted cd in the graph) and one for the conservative-inorder-branch methodology (denoted ino). All measurements are made assuming an inter-core communication latency of 4 cycles. The graph clearly indicates that this recovery policy has significant effect on the exploitable parallelism. However, the effect is far greater on single-ROB processors as compared to CMPs. Just as before, the diminishing returns remain the same and 16-cores seems to nearly saturate the exploitable parallelism using the described threading algorithm.

As an interesting aside, the graph reveals that large monolithic instruction windows often do not outperform their smaller counterparts. Given the model, this effect is quite intuitive. In the multi-core processors, since different threads of control can be executing on the different cores, a single branch misprediction is less devastating since all but one window can continue execution unhindered. Conversely, for single-ROB machines, a single misprediction causes the processor to restart execution of all subsequent instructions. Its natural that extending the window size provides very little additional parallelism due to branch mispredictions. It is worth noting, however, that in the presence of long-latency operations, the benefits of large monolithic windows over small monolithic windows would be clear. However, even in these cases, it is likely that CMP performance will still be less sensitive to individual branch mispredictions.

IV. CMP Scalability

The previous section illustrated that CMPs show remarkable potential in tapping the available parallelism in single-threaded applications. This potential remained even in the presence of inter-core communication delays and realistic branch recovery models. The experiments, however, also revealed that the utility of additional processor cores rapidly diminishes after 8 cores and is effectively insignificant beyond 16 cores if a parallelization strategy emerges with performance similar to the one presented.

Given these scalability properties, the outlook for CMPs continuing the exponential growth in single-threaded processor performance appears *extremely* near-term. If, for example, we consider that the number of transistors on a single chip will double every 18 months, and that current

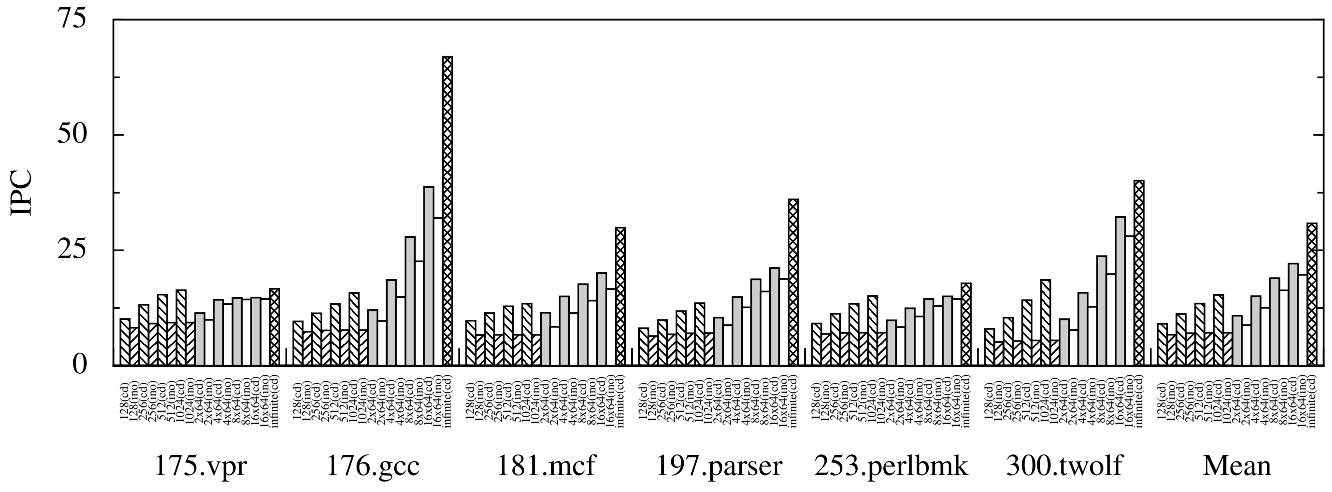

Fig. 4. Comparison of Parallelism Exploitable by Branch Types

generation chips possess two cores, we will have the technology to build a 16 way CMP in only 4.5 years provided all additional transistors are dedicated to additional processor cores. However, Sun's Niagara processor [36] may provide a 16-way CMP solution far sooner. Considering that typical systems will optimistically be running *at most* 2 to 4 compute intensive applications, in 6-8 years additional processor cores provided by technology scaling will not translate to single-threaded application performance using straight-forward application parallelization. Due to the exponential scaling, if the estimates in this paper are off by even a factor of 2 to 4, the community only has an additional 1.5 to 3 years to identify uses for additional cores to accelerate single-threaded applications or to migrate the predominant programming methodology to parallel programming.

If viewed from performance perspective, rather than integration density, the results in the previous section show that CMPs can offer a 200% speedup for single-threaded, general-purpose applications relative to a current generation uniprocessor. This corresponds to doubling performance 1.5 times. Assuming performance is also to double once every 18 months, given no clock rate increases, CMPs only provide for approximately the next two years worth of performance improvement. Process technology enhancements that improve clock rate will hopefully contribute significant performance, but even from this perspective, it seems unlikely that application parallelization will carry performance scaling beyond the next 6-8 years.

Given this forecast, the balance of this section proposes strategies for extending the performance scalability of typical applications using CMPs. The author's believe, given the delay between technology development and adoption, these research avenues should be explored now.

A. Baseline Performance

While Figure 4 shows that CMPs offer a 200% speedup potential over current generation single-core processors, the graph also estimates the IPC obtained by current generation processors to be 6.7. While this may be the parallelism available within a window of 128 instructions, modern processors typically obtain an IPC of between 1 and 2 due to real-world constraints such as long memory latencies, branch misprediction penalties due to deep pipelines, and other static or dynamic scheduling constraints. Additional processor cores may be helpful in mitigating these real-world constraints boosting the performance of single-threaded applications.

Speculative pre-computation [17], [21], [34], [39], [40], [41] is one class of techniques attacking the problem from this perspective. These techniques use additional threads, and consequently processor cores, to perform pre-fetching, warm branch predictors, or otherwise aid the primary processor core to more efficiently execute the application.

With the availability of many additional cores, these techniques can be extended to more aggressively assist the group of cores executing the application. Cores can be dedicated to performing aggressive dynamic, adaptive recompilation. Alternatively, cores can be dedicated to predicting data aliasing between parallel threads and providing hints about whether to synchronize or proceed speculatively to the cores executing those threads. Ultimately, there are limitless possibilities for using additional cores to prevent resource or data contention between cores and to provide each core a constant stream of the data and instructions needed for efficient execution. The question will be how many cores are needed before the returns are no longer worth the effort.

B. Transforming Dependences

While the techniques discussed in the previous section may dramatically improve the performance of applications, ultimately the solution is not scalable since they can only harness parallelism *that already exists*. It may be possible, however, to transform programs so they contain more intrinsic parallelism. For example, scientific application parallelism strategies often leverage privatization [4] to enhance parallelism. Basic transformations such as privatization, accumulator expansion, or other techniques which expose the order agnostic nature of many algorithms may create more opportunities for parallelism. The transformations are not limited to algorithms, but also include data structures. Consider a linked-list search for a particular datum. Having a single entry into the list fundamentally serializes the search operation. However, maintaining two or more entry points into the list offers the opportunities for parallelism. More balanced list regions, bounded by the various entry points, offer more benefit when the search is parallelized. Consequently, such algorithm/data structure modifications create new opportunities for helper threads (like those in the described in the previous section) to enhance benefits due to parallelization.

Dependence transformation is not limited to data dependences (both true and false), but also includes resource dependences. The decoupled software pipelining (DSWP) [23], [29] technique parallelizes applications by identifying data dependence recurrences within loops and scheduling different recurrences to different threads. In practice, this often manifests itself as a "producer" thread computing the loop critical path with one or more "consumer" threads processing the data produced by the critical path in parallel. The code transformation more effectively uses the scheduling resources of a single processor. Instructions stalled due to insufficient scheduling window resources are often *also* data dependent on the stalled instruction. This structure allows the critical-path thread to rapidly produce data for processing creating DOALL like parallelism for the consumers to leverage. Memory latency stalls, branch misprediction stalls, or other hiccups in the consumer thread pipelines do not prevent the producer from creating more parallelism and do not prevent other consumers from continuing processing. While DSWP is one such technique, enhancements to DSWP and other techniques which leverage similar insight may offer more scalability for single-threaded application performance.

C. Value Speculation

In addition to transforming algorithms and data structures, Lipasti and Shen [20] observed that *value prediction* offers opportunities to exceed the data-flow limit to parallelism. While the original paper suggested using value prediction hardware similar to branch predictors and leveraged the newly exposed parallelism using conventional dynamic scheduling, value speculation, in general, creates many opportunities. In the context of single-threaded application performance scalability on CMP architectures, value speculation can be used to parallelize applications, at the thread level, beyond the data-flow limit. TLS architectures, for example, already leverage value speculation. Indeterminate memory access across two threads are speculated by assuming no two accesses conflict. TLS threading strategies often also speculate the live-outs of a loop iteration to allow parallel execution of future iterations [22].

In addition to using value speculation to enhance thread-level parallelism, unused cores in CMP can be used to *generate* value predictions. The speculative values produced can be used to drive traditional dynamically scheduled processors [40] or can be used to enhance threading as described in the previous paragraph [41].

With an abundance of processor cores, the opportunities for aggressive value speculation are endless. Zilles and Sohi [41] illustrated some of the possibilities however extensions of this technique, new applications of the predicted values, and new value speculation techniques will probably all be important to continuing the scalability of CMPs for single-threaded applications.

D. New Programming Models

In addition to the compiler/architecture techniques discussed earlier, new programming models and tools to aid programmers in writing threaded applications present many opportunities for enhancing the scalability of CMPs for common general-purpose applications. Many parallel programming models already exist [8], [33], however, the lack of wide-spread adoption (few programs outside of GUI and client/server applications possess multiple threads) indicates that the improved performance offered by these models is inadequate given the development and testing costs incurred when using them.

Programming models which easily expose to the user the cost/benefit trade-off of threading, as well as those which reduce the learning curve and improve the testability of parallel programs will most likely have profound effect on the how programs are written, especially with multiprocessor systems becoming standard for both desktop and server computers. Transactional memories [28], which are related to TLS architectures, are one such model which may promote parallel programming.

Tools which highlight potential regions of parallelism also offer great opportunities. For example, the TraceVis [31] tool allows users to view the dynamic execution of a program. While the tool was designed primarily to visualize the effects of aggressive single-core processors, the tool has also been used to visualize the execution of a program running with master/slave speculative parallelization [41], and the tool supports generalized multi-core traces. Rather than using the tool with traces generated from a particular processor or simulator, TraceVis or a similar tool could be used to visualize dependences statically obtained by a compiler or those collected by a profiler. Visualization of the dependences and the available parallelism may offer programmers insight into what regions of their code naturally form threads and which regions are bottlenecks for parallel execution. With this information, manually threading a program or modifying algorithms to make them more parallel can be targeted to the regions where the effort is likely to yield significant gains. While such tools do not automatically provide CMPs with parallel workloads, they may offer the smoothest migration path between current-generation single-threaded processors and the CMPs of tomorrow.

V. Conclusion

Processor manufacturers have begun to offer chip multiprocessors instead of conventional single-core solutions in an effort to ease the design process and increase the potential performance improvement per transistor and per unit of power. These solutions naturally accelerate throughput based applications and improve system performance for multi-programmed machines with several concurrent tasks. These processors, however, do not directly offer any performance improvement to conventional, single-threaded applications.

To address this problem, the research community has been trying to find ways to parallelize (both speculatively and non-speculatively) single-threaded applications. This paper showed that it is likely that we achieve as much as a 200% speedup by parallelizing unmodified applications and also showed that it is likely that processors with at most 8-16 cores will be needed to unlock this parallelism. Given this forecast and Moore's law, this paper observed that parallelizations that respect the original programs dependences will provide for performance scalability for *only* the next 6-8 years provided all efforts are completely successful. Otherwise, the timeline is much shorter.

Given the rate of adoption of new technologies and new programming methodologies, this paper advocates exploring methodologies that will provide for performance scaling beyond the next 6-8 years. We suggested various possible opportunities including parallelization techniques to mitigate the growing disparity between processor and memory performance, techniques to mitigate the effects of branch misprediction, techniques which transform program dependences to create more intrinsic parallelism, value speculation techniques that ignore program dependences further expanding the available parallelism, and new programming methodologies and tools that will aid programmers in creating parallel programs. As the paper suggests, these techniques need not be difficult to implement, and the community has explored some such techniques. Despite these existing efforts, the community as a whole should embrace these ideas and thoroughly explore the space of designs that will be possible 8 or more years in the future.

References

[1] Advanced Micro Devices, Inc., "Multi-core processors – the next evolution in computing," Web site: http://multicore.amd.com/WhitePapers/Multi-Core_Processors_WhitePaper.pdf," White Paper, 2005.

[2] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi, "Dynamic dependency analysis of ordinary programs," in *Proceedings of the 19th International Symposium on Computer Architecture (ISCA-19)*, May 1992.

[3] U. Banerjee, *Loop Parallelization*. Boston, MA: Kluwer Academic Publishers, 1994.

[4] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh, "Automatic generation of nested, fork-join parallelism," *Journal of Supercomputing*, pp. 71–88, 1989.

[5] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proceedings of the 2004 International Symposium on Computer Architecture (ISCA)*, 2004 2004, pp. 76–89.

[6] Y. Chou, J. Fung, and J. P. Shen, "Reducing branch misprediction penalties via dynamic control independence detection," in *Proceedings of the 13th international conference on Supercomputing*. New York, NY, USA: ACM Press, 1999, pp. 109–118.

[7] A. Cristal, O. J. Santana, M. Valero, and J. F. Martinez, "Toward kilo-instruction processors," *ACM Transactions on Architecture and Code Optimization*, vol. 1, no. 4, pp. 389–417, 2004.

[8] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 21, no. 9, pp. 948–960, September 1972.

[9] M. Galluzzi, V. Puente, A. Cristal, R. Beivide, J. &#193;ngel Gregorio, and M. Valero, "A first glance at kilo-instruction based multiprocessors," in *CF'04: Proceedings of the first conference on Computing Frontiers*. New York, NY, USA: ACM Press, 2004, pp. 212–221.

[10] A. Gandhi, H. Akkary, and S. T. Srinivasan, "Reducing branch misprediction penalty via selective branch recovery," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004, pp. 254–264.

[11] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, "The stanford hydra cmp," *IEEE Micro*, vol. 20, no. 2, pp. 71–84, 2000.

[12] J. J. Hwang, Y. C. Chow, F. D. Angers, and C. Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal of Computing*, vol. 18, pp. 244–269, 1989.

[13] Intel Corporation, "A new era of architectural innovation arrives with Intel dual-core processors," *Technology@Intel Magazine*, pp. 1–11, 2005.

[14] M. Iyer, C. Ashok, J. Stone, N. Vachharajani, D. A. Connors, and M. Vachharajani, "Finding parallelism for future epic machines," in *Proceedings of the 4th Workshop on Explicitly Parallel Instruction Computing Techniques*, March 2005.

[15] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Min-cut program decomposition for thread-level speculation," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004, pp. 59–70.

[16] R. Kalla, B. Sinharoy, and J. M. Tendler, "IBM Power5 chip: a dual-core multithreaded processor," *IEEE Micro*, vol. 24, no. 2, pp. 40–47, March 2004.

[17] D. Kim, S. S. Liao, P. H. Wan, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, "Physical experimentation with prefetching helper threads on Intel's Hyperthreaded

processors," in *Proceedings of the 2004 Annual Conference on Code Generation and Optimization (CGO-3)*, March 2004, pp. 27–38.

[18] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 46–57.

[19] H. H. Lee, Y. Wu, and G. Tyson, "Quantifying instruction-level parallelism limits on an epic architecture," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2000, pp. 21–27.

[20] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 226–237.

[21] C.-K. Luk, "Tolerating memory latnecy through software-controlled pre-execution in simultaneous multithreading processors," in *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[22] P. Marcuello and A. Gonzalez, "Clustered speculative multithreaded processors," in *ICS '99: Proceedings of the 13th international conference on Supercomputing*. New York, NY, USA: ACM Press, 1999, pp. 365–372.

[23] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, November 2005.

[24] S. J. Patel, Web site: http://courses.ece.uiuc.edu/ece512/Lectures/lecture1.pdf.

[25] 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*, vol. 27, no. 1, pp. 31–34, 1999.

[26] M. K. Prabhu and K. Olukotun, "Using thread-level speculation to simplify manual parallelization," in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM Press, 2003, pp. 1–12.

[27] ——, "Exposing speculative thread parallelism in spec2000," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM Press, 2005, pp. 142–152.

[28] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005, pp. 494–505.

[29] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, September 2004, pp. 177–188.

[30] P. Ranganathan and N. P. Jouppi, "The relative importance of memory latency, bandwidth, and branch limits to performance," in *Proceedings of the Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[31] J. E. Roberts, "TraceVis: An execution visualization tool," Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, July 2004.

[32] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. P. Shen, "Coming challenges in microarchitecture and architecture," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 325–340, 2001.

[33] H. Shan and J. P. Singh, "A comparison of MPI, SHMEM and cache-coherent shared address space programming models on the sgi origin2000," in *Proceedings of the 13th International Conference on Supercomputing*. New York, NY, USA: ACM Press, 1999, pp. 329–338.

[34] G. S. Sohi and A. Roth, "Speculative multithreaded processors," *IEEE Computer*, April 2001.

[35] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000, pp. 1–12.

[36] Sun Microsystems, Inc., "Introduction to throughput computing, White Paper," 2003.

[37] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 176–188.

[38] ——, "Limits of instruction-level parallelism," DEC WRL, Tech. Rep. 93/6, November 1993.

[39] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen, "Memory latency-tolerance approaches for Itanium processors: Out-of-order execution vs speculative precomputation," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, February 2002, pp. 187–196.

[40] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[41] ——, "Master/slave speculative parallelization," in *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.