Asymmetric Chip Multiprocessors: Balancing Hardware Efficiency and Programmer Efficiency

M. Aater Suleman†, Yale N. Patt†
Eric Sprangle‡, Anwar Rohillah‡, Anwar Ghuloum‡, Doug Carmean‡

†High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin

‡Intel Corporation

TR-HPS-2007-001
February, 2007

This page is intentionally left blank

Asymmetric Chip Multiprocessors: Balancing Hardware Efficiency and Programmer Efficiency

M. Aater Suleman, Yale Patt, University of Texas at Austin

Eric Sprangle, Anwar Rohillah, Anwar Ghuloum, Doug Carmean, Intel Corporation

Abstract

Chip Multiprocessors are becoming common as the cost of increasing chip power begins to limit single core performance. The most power efficient CMP consists of low power in-order cores. However, performance on such a processor is low unless the workload is nearly completely parallelized, which depending on the workload can be impossible or require significant programmer effort.

This paper argues that the programmer effort required to parallelize an application can be reduced if the underlying architecture promises faster execution of the serial portion of an application. In such a case, programmers can parallelize only the easier-to-parallelize portions of the application and rely on the hardware to run the serial portion faster.

We make a case for an architecture which contains one high performance out-of-order processor and multiple low performance in-order processors. We call it an Asymmetric Chip Multiprocessor (ACMP). Although the out-of-order core in the ACMP makes it less power efficient, it enables the ACMP to produce higher performance gains with less programmer effort.

1. Introduction: The Power Limited Era

CPU architecture is currently transitioning from an area limited to power limited era due to silicon processing trends. Every silicon process generation shrinks linear dimensions by 30%, which has the following implications [22]:

1. Twice the transistors fit in the same die area
2. Capacitance of each transistor shrinks ~30%
3. Voltage decreases ~10%
4. Switching time of a transistor decreases ~30%

Power = CV^2F, so every process step the power scales by relative # of transistors * capacitance per transistor * relative V^2 * relative frequency, which is $2 * 0.7 * 0.9^2 * 1/.7 = 1.6$x power per silicon process generation. We can argue that voltage drops by slightly more that 10% per generation or capacitance drops by slightly more than 30%, but the conclusion that power is increasing by about 1.6x every two years is a solid conclusion. To make matters worse, power supply and dissipation cost increases non-linearly with power to the point that 300 watts is the highest economically practical design point for a chip.

After studying power consumption in traditional out-of-order processors, it quickly becomes clear that the hardware and associated power involved with uncovering parallelism must be addressed. There are many techniques for "explicitly" exposing the parallelism in the code stream so the hardware is not in charge of uncovering it. These techniques include very long vectors, threading, VLIW, etc, and each of these techniques has strengths and weaknesses depending on the type of workloads that are intended to be run. In particular, threading is a useful technique for tolerating cache misses, and many of the applications that we believe will be important in the future, like graphics rendering, media processing, etc, have poor caching behavior. Therefore we believe that threading is an appropriate technique for explicitly exposing parallelism. In addition to threading, we believe that wider SIMD could be another important technique for exposing parallelism, but we do not explore that technique in this paper.

Industry is currently taking the first steps towards adapting to the power constrained era. IPC scales roughly as the square-root of chip area [20], and power scales linear with chip area. Rather than further increasing IPC, recent designs like the Core2Duo, tile multiple traditional processors on a chip, trading some amount of single thread performance for multi-thread performance. Some designs, like Niagara, are even more radical, removing out-of-order hardware altogether and trading even more single thread performance for more multi-thread performance.

2. Architecting for Threaded Workloads

One might point out that in an area limited era, traditional cores have attempted to maximize perf/mm^2, implying the core is implicitly somewhat optimized for performance/Watt), and therefore ask "why would a core optimized for threaded workloads have a different architecture?" The reason is that traditional, single-thread optimized, cores, targeted for general purpose applications, have added features that give more than 1% performance for 3% power (1:3). 1:3 is the breakeven point for voltage scaling – if you decrease the voltage by 1%, you decrease frequency by 1%, and therefore since $P = CV^2F$, power decreases by 3% ($P = CV^2F$, $F =V$, $P = CV^3$ derivative of $V^3 = 3V^2$). Therefore any idea that improves performance by less than 1:3 is worse than simply increasing voltage, and should be rejected. In reality, traditional cores only consider ideas that provide better performance per power than 1:3, since they need to justify the engineering effort and guard against the inevitable power growth and performance degradation as ideas go from concept to reality.

For a multi-thread optimized architecture, the breakeven point is much more aggressive – 1% performance for only 1% power (not 3% power). This is because when optimizing for performance on applications that can be threaded with explicit parallelism, we can double performance by doubling the number of cores, which to a first order only doubles the power. So any feature that costs more than 1% power for 1% performance should be discarded.

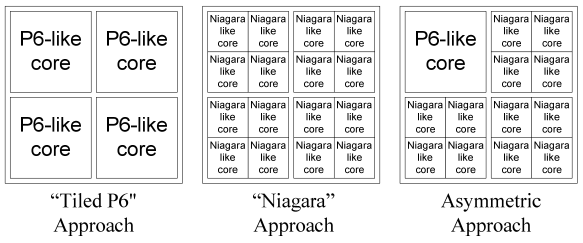

Figure 1: *CMP Architectures*

CMP	Tiled-P6 approach	Niagara approach	ACMP
P6-like cores	4	0	1
Niagara-like cores	0	16	12

Table 1: *Baseline CMP configurations.*

Core	Single P6-like Core	Single Niagara Core
Frequency	3 GHz	3 GHz
Issue-width	4	2
SMT	none	4 thread, aggressive round robin
Approximate IPC	2.2	1.4 (four threads) 0.7 (single thread)
Relative Size	1	0.25
Relative Power (Size * Freq)	1	0.25
Approximate Relative Perf / Power	1	2.5 (four threads) 1.3 (single thread)

Table 2: *Configuration of cores. Approximate IPCs come from simulation on our workloads. Relative sizes are estimated by comparing P6 to P5 with growth factors to comprehend the addition of threading and 64 bit virtual addressing. Relative power is estimated as the product of frequency and size. Relative performance/power is normalized to a single P6-like core.*

2.1. The "Tiled P6" Approach

As mentioned in Section 1, the first step towards targeting multi-threaded workloads is replicating traditional, single-thread optimized cores, for example Core2Duo, Core2Quad, AMD Opteron, and IBM Power 5 [6]. For this paper, we will call this the "Tiled P6" approach. The "Tiled P6" approach has 2 main benefits:

1. It achieves competitive performance on traditional, single-threaded workloads since each of the processors is well optimized for single threaded performance.
2. The approach requires lower design effort since the design is based on existing traditional cores.

2.2. The "Niagara" Approach

As the workload target focuses more heavily on threaded workloads, it becomes attractive to tile a core that removes the out-of-order hardware and instead exploits the threads for parallelism. Niagara is a good example of this type of architecture. In a Niagara core, 4 separate threads are picked round-robin to be issued down a relatively shallow, 5 stage in-order pipeline. If a thread stalls for any reason (for example cache miss or divide operation), the front end picker picks from the non-stalled threads in a round-robin fashion.

Rather than extracting parallelism using hardware techniques like out-of-order execution, Niagara core architecture relies on threads to create explicit parallelism. Removing out-of-order hardware and still achieving high IPC can yield substantial power efficiency improvements (~ 2.5x), but only if the thread level parallelism exists. This efficiency improvement comes at the expense of substantially lower single thread performance (on average our simulations show about 1/3rd the performance), so even a short serial bottleneck can dramatically increase overall execution time (Amdahl's law).

2.3. Asymmetric Chip Multiprocessors

Performance of an application on a given CMP architecture is dependent on its degree of parallelization. If we assume a single Niagara core is 1/4th the area while achieving 2/3rds the performance when running 4 threads, then the tiled Niagara approach can achieve higher performance vs. the tiled P6 approach on applications that are 100% parallelized. On the other extreme, if the application is not at all parallelized, the out-of-order hardware of the P6 core will help the core run the application about three times as fast.

The asymmetric chip multiprocessor (ACMP) attempts to give the best of both worlds by tiling a single P6-like core and many Niagara cores on the same chip and having all the cores run the same ISA. The ACMP can run the parallel portion of the application on the Niagara cores while running the serial portion on the faster P6. With an ACMP, the overall performance is somewhat less dependent on the percentage of parallel code. On the other hand, if a workload or application is completely parallelized, it gives up some performance because single P6-like core is getting 3/8th the throughput of the 4 Niagara cores that it replaces. In this way, an ACMP can help increase software efficiency at the expense of hardware efficiency because an ACMP can run the harder or impossible to parallelize parts of the code on the faster P6-like cores.

Note that an ACMP architecture can consist of more than 2 types of processors, with each processor type optimized for a different class of workloads. For this paper, we focus on a combination of in-order cores and out-of-order cores as we believe these cores offer the most significant tradeoff between power/performance and parallelism type [10].

2.4. What is power in the context of an ACMP?

When optimizing for performance/power, it's worth discussing exactly what we mean by power with respect to an ACMP. Will both types of cores run at the same time, or will only one type of core run at a time? If only one type of core will ever run, clock gating circuits can engage so the resulting power is the maximum of the power consumed by each core type, rather than the sum of the power.

It is hard to predict how applications will evolve to exploit the performance of an ACMP. Perhaps the closest existing analogy to an ACMP system is a PC with a CPU and a GPU, where the CPU is optimized more for single threaded performance and the GPU is optimized for the highly parallel workload of graphics rendering. In this system, most performance critical workloads, like advanced 3D games keep both the CPU and GPU fully utilized. We believe that if asymmetric chip multiprocessors become common, many applications will successfully utilize the entire compute resources and therefore the "power" should be measured as the *sum* of both the P6 core and the Niagara cores. In our simulations, both the P6 cores and the Niagara cores are used when running parallel phases of the application.

3. Background and Related Work

Asymmetric chip multiprocessors have been proposed and evaluated by several researchers. We discuss some of the contributions that are most relevant to our work.

Morad et al [2] propose an asymmetric processor that consists of one powerful core and an array of small cores. Their conclusions are similar to our conclusions presented in Figure 2 which describes performance vs. degree of parallelization, however they evaluated the performance of the processor on only one workload with a fixed degree of parallelism (0.75). Our paper shows the relation between speedup on an asymmetric processor and the degree of parallelism, which is a function of programmer effort.

Grochowski et al. also study a CMP configuration with one large core and an array of small cores [4][5], focused on optimizing the energy consumed per instruction.

Kumar et al. [7-11] focused on improving power and throughput of multi-programmed workloads. Their proposed architecture includes a combination of mediocre and powerful cores, targeting multiple, single-threaded applications running at the same time. While we believe multi-programmed workloads are important, especially in server environments, we focus on improving the performance of applications running in isolation, a usage model more relevant to desktop PC environments.

In [19], Ipek et al. show how a reconfigurable processor architecture can provide better performance on applications at intermediate stages during a parallelization effort. Balakrishnan et al. have also studied the performance and scalability of commercial workloads on asymmetric chip multiprocessors [1]. Similarly work has been done to improve scheduling of threads on asymmetric chip multiprocessors [18].

4. Contributions of this Paper

A key insight of this paper is that an ACMP trades off some amount of theoretical peak throughput for reduced programmer effort and that the percentage reduction in peak throughput decreases as the transistor budget increases.

Most previous CMP studies have focused on multi-programmed workloads. Our studies focus on existing applications which we parallelized and simulated on our CMP configurations, accurately modeling the thread management overheads, cache interactions between the threads running in parallel as well as the thread synchronization latencies.

This paper also attempts to understand some of the challenges associated with parallelizing applications, and shows how the effort associated with these challenges influences the performance of the various CMP architectures studied.

5. Degree of Parallelism

We define the "degree of parallelism" as the percentage of dynamic instructions in parallel regions of code. For simplicity, we will assume that parallel regions of code are infinitely parallel, while non-parallel regions have exactly zero parallelism. In reality, software is rarely so biased. Parallel regions of code are never infinitely parallel, and synchronization overheads, thread spawning overheads etc. can reduce or eliminate the speedup potential from parallel threads.

This simple parallel/not parallel view of the world is useful in explaining tradeoffs between the ACMP, the Niagara approach and the tiled-P6 approach. Using our performance rules of thumb from Figure 1, we plot the speedup of ACMP, Tiled-P6 and Niagara over a single P6 type core, versus the applications degree of parallelism.

When the degree of parallelism is low or non-existent, the overall performance equals the performance of one of the cores in the CMP. Both the ACMP and tiled-P6 architectures achieve about 3 times the performance of the Niagara architecture because they both include single thread optimized cores running at the same frequency of Niagara core, and we assume the P6 cores achieve three times the IPC of the Niagara core when the Niagara core has only one active thread.

At some point, there is enough parallelism that the throughput advantage of the Niagara approach offsets the single threaded performance disadvantage. For the configuration of 16 Niagara cores vs. 4 P6 cores, when the degree of parallelism exceeds 92%, the Niagara approach outperforms the tiled-P6 approach. When the degree of parallelism exceeds 96%, the Niagara approach outperforms the ACMP approach.

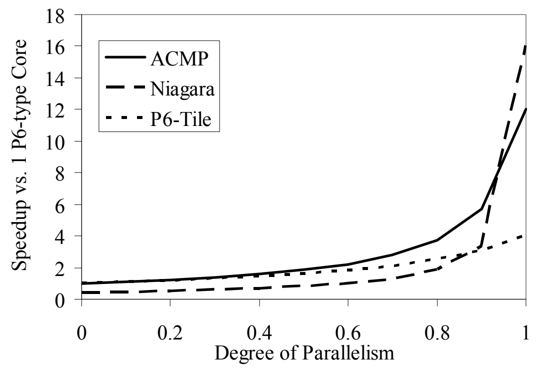

Figure 2: *Performance vs. Degree of Parallelism at a fixed peak power for different CMP architectures*

5.1. Performance of ACMP vs. alternative

As previously discussed, ACMP outperforms Niagara until the code has been 96% parallelized as shown in Figure 2 for our baseline configuration. However, the cross-over point shifts to higher and higher degrees of parallelization as the silicon process improves and we are able to put more and more processors on chip as shown in Figure 3.

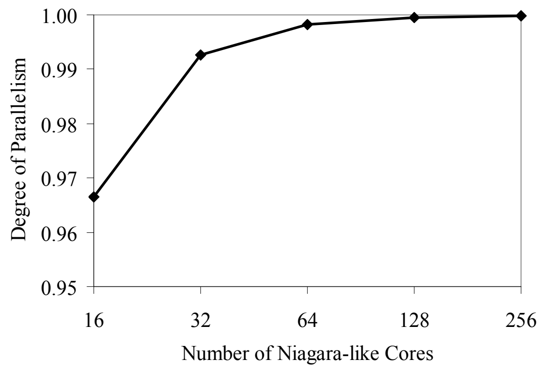

Figure 3: *Degree of parallelism needed in application for Niagara approach to outperform ACMP approach vs. die area (expressed in units of "Niagara-like cores"). Since transistor count doubles every process generation or 2 years, each doubling of Niagara cores is takes roughly 2 years.*

One process step after our baseline configuration of 16 Niagara cores vs. 12 Niagara + a single P6 cores, we double transistors and have 32 Niagara cores vs. 28 Niagara cores + a single P6 core. The peak throughput reduction of replacing 4 Niagara cores with a single P6 core is now amortized by an additional 28 Niagara cores, further increasing the degree of parallelism needed for the Niagara approach to outperform the ACMP approach.

6. Parallelizing Applications

Since the optimal CMP architecture is dependent on the degree of parallelism, we would like to better understand the challenges in parallelizing real applications.

6.1. Experiment Methodology

For our experiments, we studied 3 CMP configurations as described in Table 1 made up of the 2 basic cores described in Table 2. Each core has its own 8-way, 32kB L1 instruction and 2 cycle, 8-way, 32kB L1 data cache backed up by a unified 10 cycle, 8-way, 256kB L2 cache. All caches have 64B lines and the L2 cache is inclusive of the L1 caches. Our P6-style core includes fairly aggressive structures, including a 128-entry out-of-order window, 16kB GShare branch predictor and perfect memory disambiguation.

In all configurations, L2 cache misses cause a miss transaction to enter an on-die, bi-directional ring interconnect. Each ring has both a 64B wide data bus and an address bus. To support coherency, miss transactions first go to a tag directory that has a copy of all the tags in all the L2 caches. If a different cores' L2 cache has the line, a request is sent to that core to forward the line to the requesting core, otherwise the line is read from DRAM. For all configurations, we simulated 40 GB/sec of DRAM bandwidth with 70ns latency.

Our benchmarks were parallelized using OpenMP, and compiled to x86 using the Intel C Compiler. OpenMP allows the programmer to parallelize serial code by inserting *`pragma`* compiler directives, or assertions about parallelism, around loops and code blocks. The most common usage scenario for OpenMP is one in which loops that are known to be parallelizable are marked by the user through such `pragma` directives. OpenMP makes no effort to validate the programmer's assumptions, rather relying on programmer knowledge about the structure and access patterns in their code.

The OpenMP implementation regards the marked regions of parallelism as *quanta* of parallel work to be scheduled at run-time. For example, a loop of 1000 iterations that is marked as parallel implicitly creates 1000 quanta of work. These quanta can be treated as individual threads or, in practice, as work units that *worker* threads will execute. This latter model is commonly referred to as a *work-queuing* model of parallelism.

Practically, the work quanta are likely to be created at some coarser granularity than a single loop iteration because of the overheads of parallelism. For example, if a workload is relatively balanced across the work quanta, the run-time system is motivated to minimize total parallelism overhead by creating the minimal number of threads required to keep the cores busy. Typically, this number will be equivalent to the hardware core or thread count. Relatively imbalanced or unpredictable execution times per loop iteration will motivate the creation of more (smaller) work quanta to allow for addition quanta to schedule onto threads that complete their work earlier than others.

The overhead of thread creation is a key motivator for work-queue models. Thread creation is a relatively expensive undertaking, requiring thousands of cycles to allocate stack, thread local storage, and runtime scheduling artifacts. Moreover, groupings of work quanta created in OpenMP is of a transient nature and must be synchronized upon completion. This synchronization can also be quite expensive.

A common beginner's mistake is to overwhelm the benefits of parallelism through promiscuous thread creation for small-ish units of work. In the work queuing mode, worker threads, resources for work quanta, and associated work queues are created once at the beginning of program execution. Then, at each OpenMP `pragma` for parallelism, work quanta are added to the work queues. In our experiments, we use a scalable runtime thread management library called McRT [21] to create work queues, worker threads, and work quanta for OpenMP.

All cores support a user-level MWAIT instruction. This instruction causes a thread to enter a *sleep* state until a specified memory location is accessed by a store instruction. The Niagara pickers will not pick a thread in sleep state. The McRT can use this instruction to stop thread waiting on a lock from actively spinning and consuming core execution slots, allowing the other threads to more fully utilize the pipeline.

6.2. Quantifying Parallelization Effort

As discussed in Section 5, the ACMP approach outperforms the Niagara approach at lower parallelization levels (less than about 0.96 for our baseline comparison of 16 Niagara cores vs. 12 Niagara cores plus a single P6 core). As hardware architects, we have discussed with software engineers what we believe to be trends in computer architecture, in particular the trends towards CMPs. Invariably the conversations drift towards the complexity of threading applications and the increasing effort required to achieve greater levels of parallelizability. This increasing effort is intuitive if one assumes that the software engineer will parallelize the hot kernels of an application first, and therefore the first parallelization efforts will have the biggest effects as shown in Figure 4.

We were interested in exploring some of the challenges associated with threading workloads with the hope of gaining insights into how to address the challenge through software architecture, hardware architecture, or some combination of both.

Quantifying the effort required for parallelizing applications is difficult because it is strongly dependent on many factors:

1. The applications inherent parallelization, structure and complexity.
2. The level of programmer experience, especially with parallelizing applications.
3. The programmers' understanding of the underlying hardware.
4. The programmers' familiarity with the application being parallelized.
5. The data set being run on the application.
6. The debugging, compiler and performance tuning tools available to the programmer.

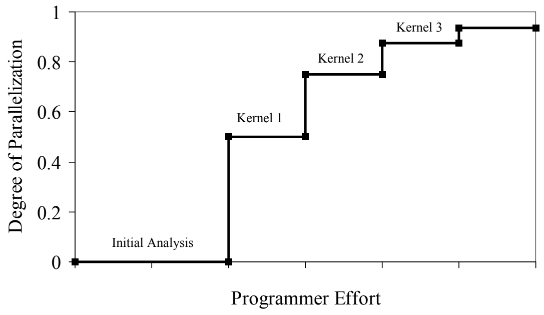

Figure 4: *Abstract representation of degree of parallelism vs. programmer effort.*

Recognizing the numerous difficulties associated with quantifying parallelization effort, it's important to focus on the general trends and understand a couple of concrete examples rather than focus on the exact effort estimates. A perfect study would lock maybe 100 professional software engineers in their rooms and have them parallelize maybe 100 different applications, recording both the parallelism and overall achieved performance versus time. As an approximation of this perfect experiment, we locked one graduate student in a room and had him parallelize 3 applications.

6.3. General Parallelization Effort Trends

Programmers generally follow a regular workflow when parallelizing applications. This entire workflow is often iteratively applied, working from hot regions of program execution to colder regions.

1. *Comprehend*: The programmer comprehends the application's control and data flow. For the domain expert, this is straightforward, though it is often the case that the domain expert is not responsible for parallelizing the applications.
2. *Analyze:* The programmer defines the regions of parallelism in the code by identifying where there are data- and control-independent regions of code. Commonly, these are loop bodies with no loop-carried data dependencies.
3. *Parallelize*: The programmer inserts OpenMP `pragma` compiler directives to identify this parallelism and measured performance.

As the application is tuned, the parallelization process becomes much more invasive with respect to modifying the original source. Moreover, at each stage, parallelism-related bugs, called *data-races,* may be introduced, which are often difficult to debug because of their non-deterministic nature.

Generally speaking, different classes of application require varying degrees of iteration through this workflow and the tuning loop. For example, a simple loop which performs embarrassingly parallel work over a very large array will likely require little tuning work. Alternatively, a loop with early exit conditions, such as a while loop, or loop-carried dependences, such as a reduction of values generated in each iteration, will require more tuning effort. We believe that the typical profile for parallelization effort is characterized by diminishing steps in performance scaling over time as colder regions of code are considered, the period and slope of each optimization phase determined by the tuning effort and programmer expertise. The general shape is illustrated in Figure 4.

7. Case Study: ART

We parallelized the SPEC2000FP application ART, which implements a neural network used to recognize objects in a thermal image. The neural network is first trained on the objects, and after training is complete, the learned images are found in the scanfield image. A window corresponding to the size of the learned objects is scanned across the scanfield image and serves as input for the neural network. The neural network attempts to match the windowed image with one of the images it has learned.

After profiling ART we found that the largest single consumer of instructions is the `train_match` function. `train_match` is called within a `while` loop which iterates until the neural network "converges" or recognizes the image to the best of the algorithms' ability.

```
while(!converge)
    train_match();
```

The `while` loop is not good candidate for parallelization since we don't know the number of iterations the loop will be executed. However, the `train_match` function itself is a much more promising candidate for parallelization because it consists of multiple `for` loops of stable and homogeneous trip count (10k iterations each for our particular input set).

```
train_match:
    loop 1 (10k iterations)
    sqrt(sum of loop 1 results)
    loop 2 (10k iterations)
    sqrt(sum of loop 2 results)
    loop 3 (10k iterations)
    sqrt(sum of loop 2 results)
    .
    .
    .
```

We first used OpenMP `pragma` directives to cause the loop to be executed as a set of parallel threads. This change did not produce speedup on our simulated CMP systems as shown in Figure 5 (the dip in performance after the first change). We measured the number of instructions executed in both the original and threaded versions of the application and found that the threaded code was executing 1.6x more instructions. This implied that the quantums of work executed by each thread was small relative to the thread creation and synchronization code. In other words, there was not enough work within each loop to offset the costs of spawning and synchronizing the individual threads.

This problem was addressed by fusing the loops and adding *barrier* operations where each loop would have needed to synchronize, as shown below:

```
train_match:
    for-loop (10k iterations)
        work from loop 1
        openMP thread barrier
        sqrt(sum of loop 1 results)
        work from loop 2
        openMP thread barrier
        sqrt(sum of loop 2 results)
        work from loop 3
        openMP thread barrier
        sqrt(sum of loop 1 results)
        .
        .
        .
```

The OpenMP barrier directives cause all the threads to stop at the barrier until all the threads have reached the barrier. After the barrier, one of the threads generates the square-root of the sum of the results generated by the parallel threads and all the threads are able to perform the work from the next loop in parallel. Note that on the ACMP, serial code is scheduled to run on the faster P6 core.

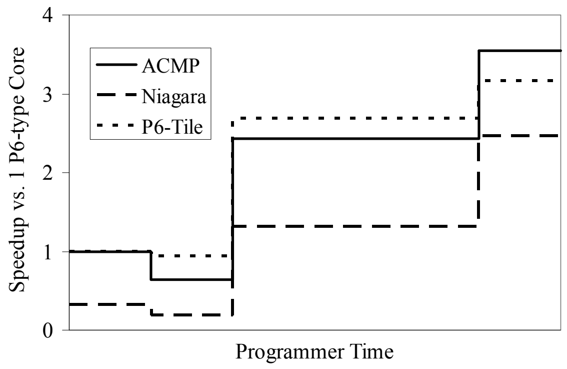

Figure 5: *Simulated speedup of parallelized ART benchmark vs. programmer effort.*

To fix this issue, the programmer had to understand both the implementation mechanisms underlying the OpenMP runtime and relative overheads of work-quanta creation and barrier synchronization. The first overhead includes thread allocation costs and work-queue scheduling algorithms that entail multiple threads contending for the work queue. The barrier synchronization, on the other hand, is a relatively well structured synchronization construct with predictable performance characteristics, especially when the work is relatively balanced between the threads, as it is in this case.

Given this new understanding, we decided to create larger parallel work quantums by parallelizing the outer loop rather than the inner loop. Again we encountered the serializing memory allocation problem where `malloc` is called within the work quantum we wish to parallelize, shown in bold below.

```
for(y=0; y < pic->rows; y++){
    q = malloc(pic->cols);
    for(x=0; x < pic->cols; x++) {
        *q = interpolate (x, sinemap….)
        q++;
    }
    output[y] = q;
}
```

This problem was fixed by moving the `malloc` out of the loop and allocating space for every iteration, shown in bold below. Note that this optimization increases the amount of virtual memory consumed by the application.

```
q2 = malloc(pic->rows * pic->cols)
for(y=0; y < pic->rows; y++){
    q = q2[y*pic->cols];
    for(x=0; x < pic->cols; x++){
        *q = interpolate (x, sinemap….)
        q++;
    }
    output[y] = q;
}
```

Our simulated performance results are shown in Figure 6. Eventually this application was parallelized to a point that both the ACMP and Niagara approach achieved nearly 5x speedup vs. the single P6-type core.

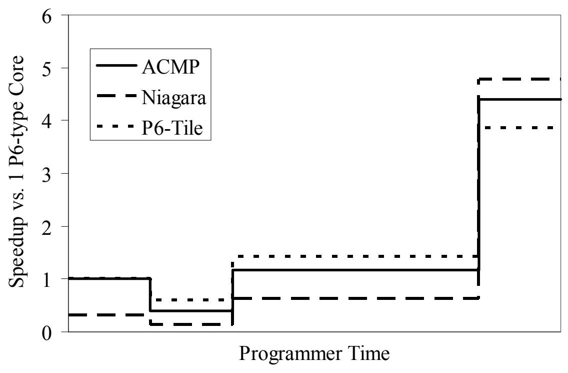

Figure 6: *Simulated speedup of Image Processing application vs. programmer effort.*

8. Case Study: Image Processing

Our next application is based on the command line image processing utility "Convert" from ImageMagick. The application reads an image file, performs the specified transformation on the image, and stores it to as specified output file.

We parallelized one of the image transformations, called "wave", which shifts each column of pixels up or down by some amount defined by a sine wave. The amplitude and frequency of the sine wave are specified by the user on the command line. The program can be split into 4 steps.

1. Reading the image from the file
2. Compute the sine wave at each column position
3. Perform the wave transformation on the image
4. Write the transformed image to a file

We profiled the code and found that most of the instructions were spent performing the wave transformation (step 3). This transformation is a loop over the rows of the image and within each row you loop over the columns of the image to transform each pixel, as show below.

```
for(x=0; x < pic->cols; x++) {
    *q = interpolate(x, sinemap….)
    *q++;
}
```

Because of the pointer arithmetic, multiple instances of the loop cannot be run in parallel. The loop needs to first be converted to using arrays, as shown below:

```
for(x=0; x < pic->cols; x++)
    q[x] = interpolate(x, sinemap….)
```

After parallelizing this loop we found that the parallel code is functionally correct but that performance degraded on our simulated CMP models, as shown in Figure 6.

After analysis, we found that there was a call to `malloc` inside the `interpolate` function (actually it was two levels deep, in the function `AcquireImagePixel`). The default McRT implementation of `malloc` has a global lock because it is calling the OS to acquire memory, and this global lock was serializing the otherwise parallel threads. The code was always allocating and freeing exactly the same amount of space and therefore it was possible to statically allocate the space, eliminating the `malloc` call. Since `AcquireImagePixel` is called in several other functions in the application, it is possible that other callers of the function actually require the `malloc` functionality to run correctly. To solve this problem, we created a duplicate copy of the function, `AcquireImagePixel2,` which did not use `malloc`, and therefore did not constrain the parallelism. Note that while there are parallel aware `malloc` packages, these incur particular overheads and do not address the *key* issue for the average programmer: Parallelizing an application requires comprehending whether any library dependences are 1) thread-safe and 2) scalable.

We continued to see performance degradation after performing this optimization. Similar to ART, we compared the instructions executed in the serial and parallel versions and realized there wasn't enough parallel work to amortize the thread creation and deletion overhead. With better understanding of McRT thread management overhead, or with use of software tools such as Thread Profiler, it may have been possible to realize that work we were parallelizing was too small before actually going through the effort of parallelizing code. However, this imposes additional burden to the programmer to learn these tools and understand the subtleties of threading runtime design.

9. Case Study: H264 Video Decoder

The H264 decoder reads an H264 compressed video stream and decodes it into video. The H264 standard promises very high compression ratio while preserving image quality. The standard divides a video picture, or *frame*, into groups of sequential rows called *slices*. These slices are further split into 16x16 pixel squares, or *macroblocks*. Each slice is read from the stream in order, decoded, and displayed or written out to a file. The H264 decoder performs the four major tasks:

1. Read the data for each slice
2. Decode each slice
3. Apply a low-pass smoothing filter
4. Write the data in the decoded picture buffer

After familiarizing ourselves with the basic video encoding standard, we profiled on a test input stream and found that most of the time was spent in the function `decode_one_frame`. At a coarse level, there are 3 nested loops within the decode process

```
for each frame in stream
    decode__frame: for each slice
        decode_slice: for each macroblock
              decode one macroblock
```

Given our understanding of the standard, we knew that there were dependencies between macroblocks in a slice, and between frames in a stream, but that slices can be processed in parallel. Researchers in the past [3] have observed that there is little benefit in decoding slices in parallel. Their observation was based on 100x100 pixel streams which do not have many slices per frame. Small video streams have few slices per frame because each slice includes a constant amount of header information, and therefore well compressed streams will have large slices. The high definition (HD) 1280x720 pixel streams that we focused on contain an order of magnitude more slices per frame. In this case, the data input set had strong influence over how to best parallelize the application, as well as the resulting level of parallelism achieved.

Before parallelizing the loop, the code needed a significant amount of cleanup to remove global variables which were being used to hold values needed within the processing of one slice, created false dependencies.

We analyzed the loop responsible for processing a slice and built the dependency graph shown in Figure 7 (Serial). For each slice, there is a serial dependency between the small header block followed by processing of the slice followed by small tail block. The header block for the following slice is dependent on the tail block of the previous slice.

After some analysis, we realized it was possible to make the tail code dependent only on the head code as shown by the dotted line in Figure 7 (Restructured). After removing this dependency it was possible to use the OpenMP `taskq` constructs (described in [15]) to reformulate the execution behave like Figure 7 (Parallelized). When a `taskq pragma` is encountered in the code, a work queue is created and worker threads are dispatched that then wait for work to be added to the work queue. The main thread continues to execute the loop and when a `task pragma` is encountered in the code, work specified within the task is added to the work queue. The key to the success of the strategy is that the body code is orders of magnitude longer than the sum of head and tail blocks.

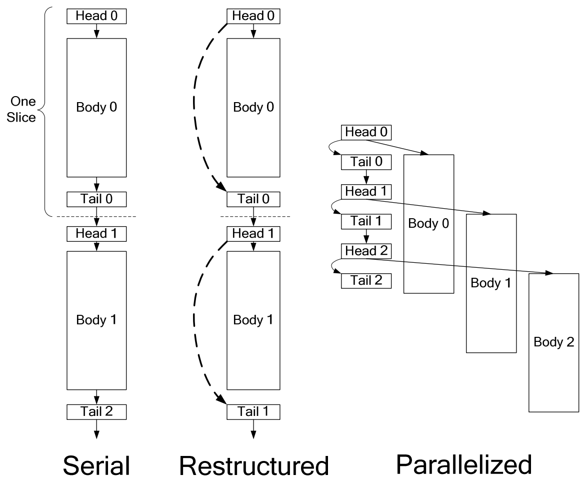

Figure 7: *H.264 decode dependency graph*

H264 at 33,000 lines is a much larger application than the image processing application at 3000 lines, which explains the long investigation phase (programmer time needed before the first optimization was implemented) as shown in Figure 8. Notice that after the initial parallelization efforts, the ACMP outperformed the Niagara approach by about 2.2x and after the final efforts, the Niagara core outperformed the ACMP core by 7%.

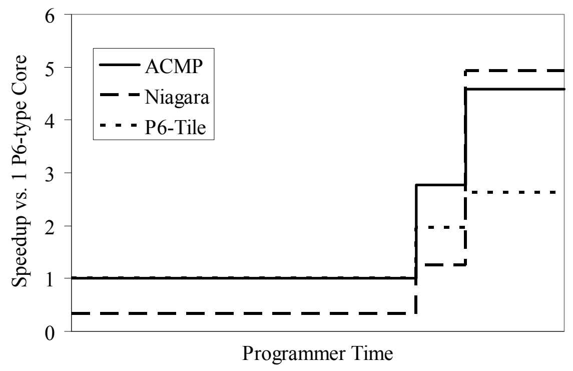

Figure 8: *Simulated speedup of H.264 decoder vs. programmer effort.*

10. Balancing Effort and Performance

All other software layers (including programming languages, libraries, and tools) being equal, it takes more programmer effort to exploit the more parallel architectures. The Niagara-like cores are the primary beneficiary of this increased performance. As observed in our simple programming experiments, the performance of the Niagara-like cores is lower than both the ACMP and P6-like cores when starting out in the parallelization process. Similarly, the performance of the Niagara-like cores eventually gains the most from additional tuning effort.

ACMP strikes a balance between these two architectural approaches, equaling the performance of P6-like cores for relatively little effort while returning nearly Niagara performance on highly parallelized workloads. While this approach potentially sacrifices performance at both extremes of programmer effort, it may be necessary until software parallelization languages and tools evolve to better support multi-core architecture.

Software technologies will significantly reduce the effort to parallelize code thoroughly. Unfortunately, such software technologies have been limited to relatively niche application spaces, such as those in High Performance Computing applications. It is unlikely that existing tools will proliferate widely as multi-core architectures scale for several reasons:

1. The HPC tools focus on languages (e.g. Fortran) that are not likely widely used in mainstream application development, where C++, C#, and Java are more widely used. For example, the use of OpenMP today effectively limits development options today to C and Fortran.
2. Mainstream software development makes extensive use of libraries and middleware that will slowly be revised to meet the needs of parallel application development. This includes rewriting these libraries for thread-safety, and scalability. The `malloc` library in the ImageMagick application is a good example of this. Earlier observations that malloc did not scale well, has led researchers to develop scalable `malloc` packages.
3. Runtimes and models for HPC are focused less on client applications (like H.264 encode and decode). This is manifested in less optimal runtimes for finer-grained parallelism (preferring to optimize for very coarse-grained thread throughput), resulting in higher threading and scheduling overheads.

In our experiences, programmer productivity is paramount for mainstream software developers. Without solving the issues above, software vendors will be slow to optimize their applications. Developing and proliferating mainstream tools will likely take ~10 years, using a well known rule-of-thumb.

From a purely software productivity-performance tradeoff point of view, these observations imply that ACMP architectures will be favored to manage the risks of committing to the “pure” approaches of the P6-like and Niagara-like architectures. Silicon process scaling trends favor the ACMP approach because even when (and *if*) effective programming tools are available ~10 years from now, including P6-like cores in an otherwise Niagara-like computing fabric is likely to be of negligible cost.

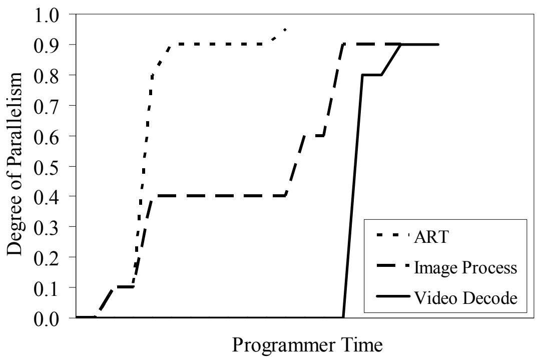

Figure 9: *Measured of degree of parallelism vs. programmer effort*

There are other factors that drive the decision to expend effort and may drive a preference for ACMP architectures. For applications based on relatively stable standards, like H.264, it is more reasonable to expect that the software developer will invest more time to optimize code. This will be well rewarded given our experiences in optimizing H.264 (see Figure 9). On the other hand, applications that evolve rapidly because of competitive pressure or applications that are likely to be highly specialized may not be worth spending significant effort in optimization. In such cases, it may be worth expending less effort (or none) to parallelize the application. The return on effort for Art and ImageMagick for example, do not justify aggressive optimization, especially if these are highly specialized (Art) or likely to be part a competitive software ecosystem (ImageMagick). An ACMP architecture that returns reasonable performance for both levels of effort is attractive to minimize the overall performance risk for the buyer.

11. Conclusion / Future Work

We have demonstrated that an ACMP outperforms Niagara unless the program is more then 95% parallelized. Therefore, unless we are able to apply the software effort required to achieve this high level of parallelism, ACMP outperforms the Niagara approach. Process scaling will further motivate ACMP vs. the Niagara approach, since the throughput opportunity cost of replacing 4 Niagara cores with a single P6 cores is amortized across more Niagara cores.

We believe there are hardware and software opportunities to reduce the parallelization effort and dramatically improve overall performance. Computer architecture trends imply that research in this area will be tremendously important in the future.

12. References

[1] Balakrishnan, S.; Ravi Rajwar; Upton, M.; Lai, K., "The impact of performance asymmetry in emerging multicore architectures," *Computer Architecture,*

2005. ISCA '05. Proceedings. 32nd International Symposium on , vol., no.pp. 506- 517, 4-8 June 2005

[2] Morad, T.Y.; Weiser, U.C.; Kolodnyt, A.; Valero, M.; Ayguade, E., "Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors," *Computer Architecture Letters, IEEE* , vol.5, no.1pp. 14- 17, Jan.-June 2006

[3] Yen-Kuang Chen, Xinmin Tian, Steven Ge, Milind Girkar, "Towards Efficient Multi-Level Threading of H.264 Encoder on Intel Hyper-Threading Architectures," *ipdps*, p. 63b, 18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Papers, 2004.

[4] Grochowski, E.; Ronen, R.; Shen, J.; Hong Wang, "Best of both latency and throughput," *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on* , vol., no.pp. 236- 243, 11-13 Oct. 2004

[5] Annavaram, M.; Grochowski, E.; Shen, J., "Mitigating Amdahl's law through EPI throttling," *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on* , vol., no.pp. 298-309, 4-8 June 2005

[6] Kalla, R.; Balaram Sinharoy; Tendler, J.M., "IBM Power5 chip: a dual-core multithreaded processor," *Micro, IEEE* , vol.24, no.2pp. 40- 47, Mar-Apr 2004

[7] Kumar, R., Farkas, K., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. 2006. Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures. *IEEE Comput. Archit. Lett.* 1, 1 (Jan. 2006), 5-8.

[8] Kumar, R., Tullsen, D. M., Jouppi, N. P., and Ranganathan, P. 2005. Heterogeneous Chip Multiprocessors. *Computer* 38, 11 (Nov. 2005), 32-38.

[9] Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st Annual international Symposium on Computer Architecture* (München, Germany, June 19 - 23, 2004).

[10] Kumar, R., Tullsen, D. M., and Jouppi, N. P. 2006. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th international Conference on Parallel Architectures and Compilation Techniques* (Seattle, Washington, USA, September 16 - 20, 2006). PACT '06. ACM Press, New York, NY, 23-32.

[11] Kumar, R.; Farkas, K.I.; Jouppi, N.P.; Ranganathan, P.; Tullsen, D.M., "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* , vol., no.pp. 81- 92, 3-5 Dec. 2003

[12] Amdahl, G. M. 2000. Validity of the single processor approach to achieving large scale computing capabilities. In *Readings in Computer Architecture*, M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. Morgan Kaufmann Publishers, San Francisco, CA, 79-81.

[13] Gustafson, J. L. 1988. Reevaluating Amdahl's law. *Commun. ACM* 31, 5 (May. 1988), 532-533.

[14] Davis, J.D.; Laudon, J.; Olukotun, K., "Maximizing CMP throughput with mediocre cores," *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on* , vol., no.pp. 51- 62, 17-21 Sept. 2005

[15] X. Tian, M. Girkar, S. Shah, D. Armstrong, E. Su, and P. Petersen. "Compiler support of the workqueuing execution model for intel smp architectures". *In Proc. of the Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 47–55, Nice, Paris, Sep 2003.

[16] Becchi, M. and Crowley, P. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd Conference on Computing Frontiers* (Ischia, Italy, May 03 - 05, 2006). CF '06. ACM Press, New York, NY, 29-40.

[17] Figueiredo, R.J.O.; Fortes, J.A.B., "Impact of heterogeneity on DSM performance," *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on* , vol., no.pp.26-35, 2000

[18] Grant, R.E.; Afsahi, A., "Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* , vol., no.pp. 8 pp.-, 25-29 April 2006

[19] Engin İpek, M. Kırman, N. Kırman, and J.F. Martínez. Accommodating workload diversity in chip multiprocessors via adaptive core fusion. In *Workshop on Complexity-effective Design*, conc. with ISCA, Boston, MA, June 2006

[20] Pollack, F. J. 1999. New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)(abstract only). In *Proceedings of the 32nd Annual ACM/IEEE international Symposium on Microarchitecture* (Haifa, Israel, November 16 - 18, 1999).

[21] Saha, H. et al., Enabling Scalability and Performance in a Large Scale Chip Multiprocessor Environment, *Submitted to Eurosys 2007.*

[22] Borkar, S. Design Challenges of Technology Scaling. *IEEE Micro.* 1999.