Mitigating Amdahl's Law Through EPI Throttling

Murali Annavaram, Ed Grochowski, John Shen
Microarchitecture Research Lab, Intel Corporation
2200 Mission College Blvd, Santa Clara, CA 95054
murali.m.annavaram@intel.com, edward.grochowski@intel.com, john.shen@intel.com

Abstract

*This paper is motivated by three recent trends in computer design. First, chip multi-processors (CMPs) with increasing numbers of CPU cores per chip are becoming common. Second, multi-threaded software that can take advantage of CMPs will soon become prevalent. Due to the nature of the algorithms, these multi-threaded programs inherently will have phases of sequential execution; Amdahl's law dictates that the speedup of such parallel programs will be limited by the sequential portion of the computation. Finally, increasing levels of on-chip integration coupled with a slowing rate of reduction in supply voltage make power consumption a first order design constraint. Given this environment, our goal is to minimize the execution times of multi-threaded programs containing nontrivial parallel and sequential phases, while keeping the CMP's total power consumption within a fixed budget. In order to mitigate the effects of Amdahl's law, in this paper we make a compelling case for varying the amount of energy expended to process instructions according to the amount of available parallelism. Using the equation, Power=Energy per instruction (EPI) * Instructions per second (IPS), we propose that during phases of limited parallelism (low IPS) the chip multi-processor will spend more EPI; similarly, during phases of higher parallelism (high IPS) the chip multi-processor will spend less EPI; in both scenarios power is fixed. We evaluate the performance benefits of an EPI throttle on an asymmetric multiprocessor (AMP) prototyped from a physical 4-way Xeon SMP server. Using a wide range of multi-threaded programs, we show a 38% wall clock speedup on an AMP compared to a standard SMP that uses the same power. We also measure the supply current on a 4-way SMP server while running the multi-threaded programs and use the measured data as input to a software simulator that implements a more flexible EPI throttle. The results from the measurement-driven simulation show performance benefits comparable to the AMP prototype. We analyze the results from both techniques, explain why and when an EPI throttle works well, and conclude with a discussion of the challenges in building practical EPI throttles.*

1. Introduction

With silicon technologies enabling billions of transistors on a single chip, the era of the Chip Multiprocessor (CMP) has arrived [4][11][15]. CMPs will soon be used across all computing domains: server, desktop and mobile. Starting from today's dual-core processors, the number of CPU cores on a chip can be expected to grow over time as a result of Moore's Law.

The ubiquitous presence of CMPs will naturally be accompanied by a proliferation of multi-threaded software. Although multi-threaded software is primarily the domain of servers today, we expect that in the near future software for desktop and mobile computers will also become multi-threaded. Due to the inherent nature of the algorithms, multi-threaded software will contain phases of sequential execution interspersed with phases of parallel execution. Amdahl's law states that the speedup possible due to parallelization will be limited by the portion of the time spent in the sequential component. For example, if a program spends 10% of its time in a sequential component, the maximum speedup achievable through parallelization is 10. Due to Amdahl's law, it is necessary to improve the performance of both the sequential as well as the parallel components of execution.

Historically, advances in silicon process technology have been accompanied by reductions in power supply voltage. Reducing supply voltage lowers chip power consumption and maintains the electric field strength within the transistors at safe levels. In recent years, however, a slowdown in the rate of reduction in supply voltage has hampered our ability to lower power consumption with new process technologies. Thus power consumption has emerged at the forefront of challenges facing the chip designer. Researchers have proposed many techniques to reduce power and energy in accordance to the software's behavior [1][5][14][16][17]. In this work our goal is to maximize the performance of a CMP while keeping its total power consumption within a fixed power budget. Unlike battery operated devices that strive to conserve energy to extend battery life, we focus on AC line powered server and desktop systems that strive to deliver maximum performance while operating just within the capabilities of the power delivery and cooling subsystems.

Due to the conflicting microarchitectural demands, it is nearly impossible to optimize both single-threaded (sequential component) and throughput (parallel component) performance within a fixed power budget. Microarchitectural techniques for reducing single-threaded execution latency, e.g. out-of-order execution, speculation, and deep pipelining, consume relatively high energy per instruction, thereby limiting the number of CPU cores that can be accommodated in a CMP for a given power budget. On the other hand, throughput performance demands many low energy cores in a CMP to exploit thread-level parallelism.

The key to designing a microprocessor that can achieve both high scalar performance and high throughput performance is to dynamically vary the amount of energy expended to process each instruction according to the amount of parallelism available in the software. In other words, if there is little parallelism, a microprocessor should expend all available energy processing a few instructions; and if there is a lot of parallelism, the microprocessor should expend less energy in processing each instruction. This relationship may be formalized as:

$$P = EPI * IPS \qquad (1)$$

where P is the fixed power budget, EPI is the average energy per retired instruction, and IPS is the aggregate number of instructions retired per second across all CPU cores.

In [9] the authors discussed four techniques for varying EPI. Table 1 summarizes the four techniques, showing the achievable range of EPI (conservative range to optimistic range), the approximate time required to vary EPI over the stated range, and the action that an EPI throttle would take to reduce energy. To accommodate software with a wide range of IPS, an equally wide range of EPI is required. As shown in the table, designers have the choice of EPI techniques that are relatively slow but provide significant EPI range, or EPI techniques that are much faster but have limited EPI range.

Method	EPI Range	Time to vary EPI	Throttle Action
Scaling voltage frequency	1:2 to 1:4	100us;ramp Vcc	Lower voltage frequency
Asymmetric cores	1:4 to 1:6	10us;migrate L2	Migrate threads between cores
Variable-size core	1:1 to 1:2	1us;fill L1	Reduce toggled capacitance
Speculation control	1:1 to 1:1.4	10ns;flush pipe	Reduce speculation

Table 1. EPI Throttling Techniques

In [9] the authors described the concept of EPI throttling. In this paper we present a comprehensive performance evaluation of EPI throttling using a combination of prototyping and measurements on physical systems. This paper makes the following contributions:

1. By using a novel combination of clock throttling and processor affinity, we prototype an asymmetric multiprocessor (AMP) using an off-the-shelf 4-way SMP. Each processor in the AMP expends varying amount of EPI based on the available thread-level parallelism.

2. We show an average of 38% wall clock speedup on a wide range of multi-threaded programs running on our physical AMP prototype compared to a standard SMP that uses the same power. To the best of our knowledge, this paper is the first to prototype an AMP and to show wall clock speedups on a physical system.

3. Our AMP prototype is limited to varying the EPI of a processor by a coarse granularity. In order to assess the impact of this limitation, we measure the supply current on a physical 4-way SMP server running multi-threaded programs. We feed the measured data as input to a software simulator that implements a more flexible EPI throttle. We show comparable speedups between the simulator and the AMP prototype, indicating that a realistic EPI throttle can deliver the performance gains in practice.

4. Using two complementary methods, we make a compelling case that an EPI throttled AMP is effective in mitigating the effects of Amdahl's law while running multi-threaded programs with non-trivial amounts of sequential component.

The rest of this paper is organized as follows. Section 2 discusses how this work differs from the previous research. Section 3 describes our experimental prototype. Section 4 describes our multi-threaded programs setup and execution. Section 5 presents the results comparing the execution times of our programs on AMP with two machine configurations that consume equal power. We analyze the reasons when and why an EPI throttle performs well. Section 6 presents the results of the current measurement and software simulation technique. Section 7 presents conclusions and areas for future work.

2. Related Work

Much research has been done on various microarchitectural techniques to reduce energy per instruction. These techniques include CMOS voltage/frequency scaling [6], asymmetric multiprocessor cores [16][17][21], variable-sized cores (also called adaptive processing) [1], and speculation control [19]. Most of this prior work has been done in

the context of running single-threaded programs with the goal of reducing energy without sacrificing too much performance. Most researchers cite an improvement on a metric such as energy*delay^2 with power and performance computed through simulations, measured activity factors, or analytical models. This paper, on the other hand, focuses on reducing the wall-clock execution times multi-threaded programs running on future CMPs using a physical prototype of an AMP.

The idea of regulating (or throttling) a processor's activities to control power and temperature has been the topic of much research and product development. Brooks and Martonosi [5] described the use of dynamic thermal management to control die temperature. The Intel® Pentium® 4 processor implements a thermal monitor to limit die temperature so that the processor and system thermal solutions may be designed according to the power envelopes of real programs rather than worst-case power viruses [13]. We use the clock throttling feature on the Pentium 4 processor in conjunction with process affinity features in Linux to prototype an AMP.

Heterogeneous multiprocessors are another area of related work. Menasce and Almeida [20] proposed the use of heterogeneous processors in supercomputers, with two different types of processors used to speed up the parallel and sequential portions of a computation. Figueiredo and Fortes [8] explored heterogeneous distributed-shared-memory multiprocessors with a few nodes with large caches designed for single-thread performance, and a larger number of nodes with smaller caches designed for multi-threaded parallelism. Morad et al. [21] presented an analysis of why an asymmetric chip multiprocessor can achieve higher performance for a given area and power budget. Li and Martinez [18] used a combination of analytical models and simulation experiments to show that parallel computing can bring significant power savings while meeting a given performance target, by choosing appropriate granularity for parallelism and judicious voltage/frequency levels. Kumar et al. [16] proposed a heterogeneous multi-core processor that delivers higher performance than a homogeneous multi-core processor when both are constrained by the same die area. Our AMP prototype is closest to the heterogeneous CMP presented in [16]. In [16] a single threaded program is switched between multiple cores on a CMP based on the heuristic that seeks to minimize energy-delay product. This paper differs in three respects: we assume the main constraint is a fixed power budget; we exclusively use multi-threaded workloads; and we measure wall clock speedup instead of relying on a metric such as energy-delay.

3. Prototyping AMP on an SMP

To evaluate the performance benefits of EPI throttling, we construct a prototype multiprocessor system that uses the Intel® Pentium® 4 Processor's clock throttle mechanism [10] to create multiple performance and power operating points. The physical AMP prototype system described in this section and the software modifications described in Section 4 are primarily intended for evaluating the benefits of EPI throttling. In practice, we expect an EPI throttle to be implemented as a hardware mechanism that operates transparently to software.

The Pentium 4 processor uses on-chip temperature sensing diodes and clock throttling to control both the temperature and power consumption of the chip. When the measured temperature exceeds a defined threshold, hardware automatically shuts off the processor clock for a brief time interval to allow the chip to cool. Extensive use of clock gating and unit power down techniques allows the power consumption to drastically drop during clock throttling. System software can also explicitly manage power consumption using the same clock throttling mechanism.

Clock throttling can be configured by writing to the IA32_CLOCK_MODULATION model specific register (MSR). Three duty cycle bits in the MSR set the duty cycle to one of the seven levels – 12.5%, 25%, 37.5%, 50%, 62.5%, 75% and 87.5%. The duty cycle can be set on a per processor basis in a multiprocessor. Note that clock throttling does not reduce the actual operating voltage or frequency of a processor. However, clock throttling has a similar effect on performance as reducing the processor frequency. Based on historical data [9] we *assume* that it is possible to design a range of CPU microarchitectures in which the power consumption may be made proportional to the square of the frequency. For instance, if we duty cycle a 2GHz processor at 50% then the processor performance is *approximately* similar to the performance of a 1GHz processor (½ the performance of 2GHz processor), and the 1GHz processor consumes roughly ¼ the power of the 2GHz processor. Thus, EPI is reduced by a factor of 2. In order to emulate an EPI throttle, we vary the duty cycle of each processor independently to create or emulate an AMP. Note that we conservatively use a square relationship between power and performance rather than an idealized cubic relationship as in CMOS voltage/frequency scaling.

In our experiment, to satisfy equation 1 for a given power budget, we assign parallel phases of a program with high aggregate IPS to low EPI processors and sequential phases with low IPS to high EPI processors. In order to make such an assignment we need the ability to specify to the OS scheduler to assign a process to a particular processor. Thus our AMP prototype relies on

both clock throttling and affinity control, which are fortunately supported by the Linux 2.4 kernel. The kernel module *p4-clockmod* provides a simple interface to configure the duty cycle for clock throttling. Once the module is loaded the superuser can simply update the file "/proc/cpufreq" file with the desired duty cycle value and within a short interval (10-100 microseconds) clock throttling will be initiated. We wrote a simple driver routine that takes as input the processor number and the desired duty cycle value and updates the *cpufreq* file.

The second Linux kernel feature used in our prototype is the process scheduling feature that allows a thread/process to be affined to a specific processor. Linux provides a command line scheduling tool, called *taskset*, which allows any running process to be scheduled to run on a specific processor. Linux also provides a C-programming interface through a set of process scheduling API, *sched_setaffinity* and *sched_getaffinity*, which allows a programmer to affine a thread to a processor. Section 4 describes how clock throttling and processor affinity features are used to run a diverse set of multi-threaded programs on the AMP prototype.

3.1   Hardware Configuration

We chose an Intel Xeon™ 4-way SMP server as our experimental machine. The Xeon processor is the server version of the Pentium 4 processor. The Xeon processors in our system operate at 2GHz with Hyper-Threading Technology disabled. Our system is populated with 4 GB of PC200 DDR and has 3 Ultra320 SCSI drives, each with 73 GB of capacity. We installed Red Hat* Enterprise Linux 3.0 OS with 2.4.21-15.ELsmp kernel.

In our experiments, we fixed the power budget to be same as the power consumed by a 1P Xeon processor running at 2GHz. Given this power budget, the base SMP can be configured into several MP configurations all of which consume roughly the same power. As mentioned earlier, based on historical data we assume that a range of CPUs may be designed in which power is proportional to the square of the duty cycle. For instance, if the power consumed by the baseline 1P/2GHz processor is 1, then 2P running at 6/8 duty cycle (effective frequency is 6/8*2GHz =1.5GHz) consume $2*(6/8)^2$=1.12. Table 2 shows the four possible configurations.

3.2   AMP Configuration

We created two AMP configurations using a combination of configurations shown in Table 2. One is a static AMP configuration where either three processors

run at 1.25GHz (5/8 duty cycle) or one processor runs at 2GHz. We use either the 1P or the 3P depending on the amount of thread-level parallelism in our programs. When the program executes sequential code, we run the sequential code on the one *fast* processor and when the program executes parallel code we run the parallel threads on the three *slow* processors. Using processor affinity we guarantee that the sequential phase is affined to the fast processor and parallel phases are affined to the slow processors. In our prototype, we assume that during sequential code execution the three slow processors can be shut off so as to consume negligible power, and similarly during parallel execution the fast processor can be shut off so as to consume negligible power. An extremely low power state may be achieved in practice through the use of sleep transistors and body bias [24]. In the static AMP setup, the duty cycle is set only once before a program run and is not altered during execution.

CPUs	Duty Cycle	Effective Freq	Normalized Total Power
1P	8/8	2GHz	1.00 (Baseline)
2P	6/8	1.5GHz	1.12
3P	5/8	1.25GHz	1.17
4P	4/8	1GHz	1.00

Table 2. MP Configurations with Similar Power

The second AMP configuration is dynamically reconfigured during program execution based on the amount of available thread-level parallelism. The parallel phases are run on either all or a subset of the four processors while the sequential phase runs on a single processor at 2GHz. For instance, if there are four available threads in a given phase we use four processors running at 1GHz (duty cycle of 4/8) and if the number of threads reduces to two then we reassign the power budget to just two processors by running them each at 1.5GHz (duty cycle of 6/8). Thus the power consumed by AMP is constant all through the program run; EPI is lower during parallel execution phases and EPI is higher during sequential execution phases.

A static AMP is desirable for those programs that rapidly transition between sequential and parallel phases as the overhead of frequently changing the duty cycle (by updating the cpufreq file) is quite significant. On the other hand, a dynamic AMP is desirable when the thread-level parallelism varies between one and four during program execution and there are only a few transitions between sequential and parallel phases. While some of the overhead is an artifact of our prototype implementation, we believe that any mechanism to vary EPI will be fundamentally slow relative to instruction execution. Hence, reconfiguration overheads should be

---

* Other names and brands may be claimed as the property of others.

taken into consideration when choosing an EPI throttling mechanism.

Some configurations in Table 2 use slightly more power than the baseline due to the coarse granularity of the duty cycle. Since some AMP configurations consume up to 17% more power, they have up to an 8% performance advantage. To eliminate this advantage and reflect constant power conditions, we have adjusted the run times on the AMP setup for all the results presented in this paper. For instance the measured AMP runtime on wupwise is 647 seconds, where 90% of the time wupwise runs on 3P/1.25GHZ and 10% of the time on 1P/2GHz. Hence, the adjusted run time is (1+0.9*0.08)*647=694 seconds. In doing this adjustment, all the MP configurations compared in this paper can be viewed as consuming the exact same fixed power.

4. Benchmarks: Setup and Tuning

We selected 13 parallel programs for this study: 9 SPEC Open MP (OMP) benchmarks [3], BLAST [2]and HMMER [12] bioinformatics programs, TPC-H [23] decision support program, and FFTW a parallel Fourier transform solver [7]. The SPEC OMP benchmarks are compiled with the Intel C++ compiler 8.0 and the Intel Fortran compiler 8.0. All other programs are compiled with gcc 3.2.3. These represent a wide range of realistic multi-threaded programs that we were able to easily set up and build from the source code. This section describes the programs and explains how we set up and run them on our AMP prototype.

4.1 SPEC OMP

The SPEC OMP benchmarks are a subset of the well known CPU2K benchmarks. These benchmarks are parallelized using the Open MP directives. We selected 9 out of the 11 benchmarks from the SPEC OMP suite: *wupwise*, *swim*, *mgrid*, *applu*, *equake*, *apsi*, *fma3d*, *art* and *ammp*. Two benchmarks, *galgel* and *gafort* were not used as they could not be run to completion in our environment. Based on the frequency of transitions between sequential and parallel regions we chose either a static AMP or a dynamic AMP configuration. Table 3 shows AMP configurations used for each of our 13 programs. For most SPEC OMP benchmarks we used static AMP configuration, where clock throttling is used to statically configure an AMP with one processor (CPU 0) at 2GHz and three processors (CPUs 1, 2 and 3) at 1.25GHz. Typically OMP parallelization directives create as many threads as there are CPUs (four in our setup). However, we hand modified all the benchmarks to use processor affinity guaranteeing that the sequential regions run only on the CPU 0 running at 2GHz and the

parallel regions run only on CPUs 1, 2 and 3 running at 1.25 GHz. Modification of the SPEC OMP benchmarks is fairly simple as we searched for the parallelization directives and added processor affinity calls at the start and end of each OMP directive. These software modifications were only necessary in our prototype implementation, while in practice we expect an EPI throttle to be implemented as a hardware mechanism that operates transparently to software.

AMP Configuration	Programs
Static AMP: 1P/2GHz or 3P/1.25GHz	wupwise, swim, mgrid, equake, fma3d, art, ammp, BLAST, HMMER
Dynamic AMP: 1P/2GHz to 4P/1GHz	applu, apsi, FFTW, TPC-H

Table 3. AMP Configurations for Programs

4.2 BLAST

Basic Local Alignment Search Tool (BLAST) is a program extensively used in bioinformatics research where researchers search for the closest match of a new protein or amino acid sequence in an existing, and continuously evolving, database of known sequences. Typically the database is built from FASTA formatted text input data using the *formatdb* program. Formatting reorganizes text data into a meta-database that enables efficient search operations. After the database is built usually a series of repeated searches are performed on the database using the *blast_all* program. The blast tool kit provides code for both *formatdb* and *blast_all*. Database formatting is a sequential operation while search operations are parallel.

We ran blast on a static AMP configuration. The code provided by BLAST uses pthreads to automatically create as many threads as there are CPUs during search. We modified the code to create only three threads during the search operation. Furthermore, we used the Linux thread scheduling API (*sched_setaffinity*) to assign the three parallel search threads to CPUs 1, 2 and 3 running at 1.25 GHz and the sequential *formatdb* operation is configured to run on the 2GHz CPU 0. In our setup we ran the *formatdb* operation on a 120MB database and ran five parallel search operations using a 2500 character sequence protein string.

4.3 HMMER

HMMER is another bioinformatics program which is similar to BLAST. HMMER, unlike BLAST, concurrently searches for multiple sequences in a database using Hidden Markov Model pattern matching.

HMMER is capable of searching text databases and hence does not need any database formatting operation. The search operation is mostly parallel, except during the initial setup and output generation phases which are sequential. Our AMP setup for HMMER is also statically configured where 1P/2GHz runs the sequential code and 3P/1.25GHz run the parallel search code.

4.4 FFTW

FFTW is a collection of programs for computing the Discrete Fourier Transform in multiple dimensions. It includes complex, real, symmetric, and parallel transforms, and can handle arbitrary array sizes efficiently. For our work we used the three-dimensional FFT solver parallelized using pthreads. The program uses data parallelism by dividing the input data arrays into as many chunks as there are processors. Each processor operates on its own chunk and the results are then merged at the end. FFTW has just four transitions between sequential and parallel phases. Furthermore, during the parallel operation there is enough data parallelism to fully saturate four CPUs. Hence, we used a dynamic AMP setup where the sequential regions run on 1P/2GHz and the parallel region run on 4P/1GHz. We identified sequential and parallel regions in the program and affined the sequential regions to run only on CPU 0. Prior to the beginning of each sequential region we set the CPU 0 duty cycle to 1 and at the end of the sequential region we reset the duty cycle to 4/8. Thus the parallel regions run on all four CPUs at duty cycle 4/8 and the sequential region runs on CPU 0 at duty cycle 1.

4.5 TPC-H

TPC-H is a decision support database benchmark. We built a 1GB TPC-H database using the Postgres DBMS [22] and tuned the setup to maximize the CPU utilization. We wrote a master driver to run all the 22 queries in TPC-H. The driver places all these queries in a query queue in random order. Initially, our driver sets the frequency of the four CPUs to 1GHz. Then it creates four threads and assigns affinities to run them on four different CPUs. Each thread then executes one query from the query queue on the thread's assigned processor. When a query completes the thread picks up the next query from the queue. If there are no more queries in the queue the thread signals to the master and exits. The master thread then increases the frequency of the remaining three threads to 1.25GHz. Similarly, when the next thread exits, the master increases the frequency of the remaining two CPUs to 1.5GHz. Finally when there is only one thread the frequency is increased to 2GHz. In this manner, the total power is kept constant over all phases.

5. Results and Analysis

Figure 1 shows the results of running the 13 programs on three machine configurations (1P/2GHz, AMP, and 4P/1GHz). All three configurations consume roughly equal power. The left bar for each program is the speedup on an AMP normalized to 1P/2GHz. The right bar is the speedup on 4P/1GHz SMP normalized to 1P/2GHz. Based on the speedups achieved, the programs may be classified into one of three categories. The five programs in the first category (ammp, applu, apsi, mgrid, and wupwise) show that a 4P/1GHz SMP performs slightly better than AMP. The six programs in the second category (art, BLAST, equake, FFTW, HMMER, and TPC-H) show that AMP achieves significantly better speedup than 1P/2GHz or 4P/1GHz. Finally, the two programs, fma3d and swim, indicate AMP and 4P/1GHz do not provide any performance benefit (or even worse performance) over a standard 1P/2GHz processor.
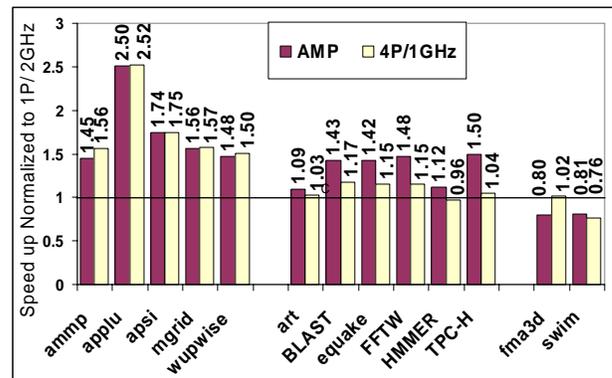

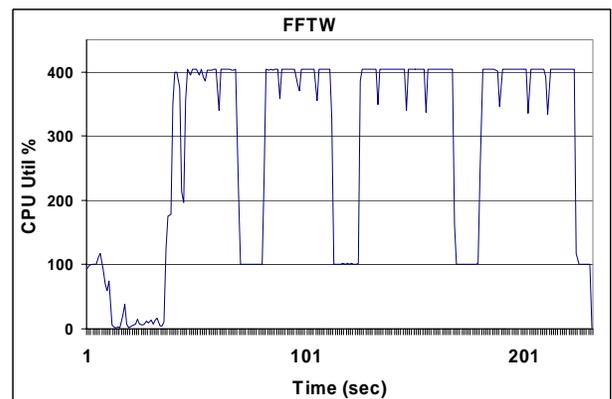**Figure 1. Speedup Normalized to 1P/2GHz**

Figure 2. CPU utilization over time for FFTW

To provide an intuitive understanding of why AMP improves performance in the second category, Figure 2 shows the measured total CPU utilization during the

AMP run of FFTW. The graph shows distinct phases of sequential and parallel execution where the CPU utilization is 100% and 400% respectively. AMP uses 1P/2GHz processor during sequential phases and uses a 4P/1GHz during parallel phases. Using a standard 4P/1GHz SMP would underutilize the power budget during sequential phases when only one processor is running. In contrast, always using a 1P/2GHz processor does not take advantage of available thread-level parallelism during parallel phases. AMP improves performance by varying the EPI according to available thread-level parallelism to continuously optimize the use of given power budget.

We now present a detailed analysis of when and why AMP works better than SMP. Figure 3 shows idealized run times (normalized to 1P/2GHz) of a 4-way SMP and an AMP along with the runtimes of the 13 programs. The X-axis shows the percentage of sequential component in the program, and the Y-axis shows the run time normalized to 1P/2GHz.
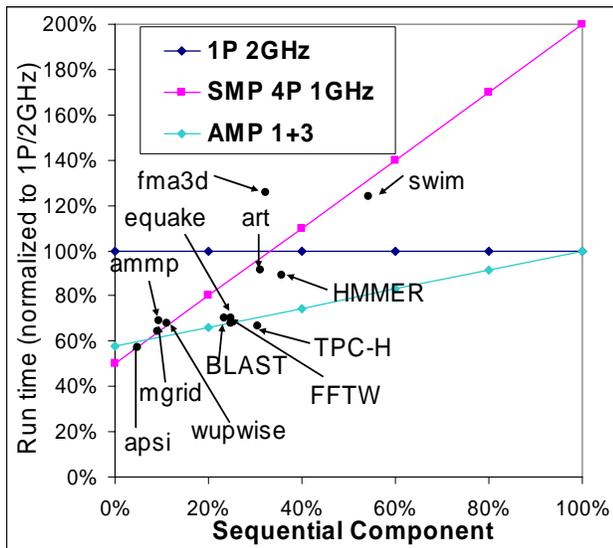

Figure 3: Ideal versus Measured Run time

As shown in the graph, when the sequential component is 0%, 4P/1GHz achieves half the execution time of 1P/2GHz because the program is completely parallelized. When the sequential component is 100%, 4P/1GHz has twice the execution time of 1P/2GHz since the multiprocessor is using only one of its four CPUs. For sequential components in between 0% and 100%, the run times may be linearly interpolated from the two endpoints because the sequential and parallel components are additive. Figure 3 also plots the run time of an idealized AMP configuration that consumes the same power: 1P/2GHz or 3P/1.25GHz (static AMP). The endpoints at 0% and 100% are about the same as the corresponding SMP configurations. However, the AMP

has the advantage that in between 0% and 100%, the AMP can optimize both the sequential and parallel phases within the fixed power budget. The idealized AMP configurations achieve lower execution times than the idealized SMP configurations for all but the trivially parallel or completely sequential programs.

Finally, Figure 3 plots the measured run times of the 12 programs (applu is omitted because its speedup is superlinear). The sequential component is determined by measuring the execution times on a 1P/2GHz and 4P/1GHz and applying the following formulas:

$$u = s + p \qquad\qquad (2)$$
$$m = 2 \cdot (s + p/4) \qquad\qquad (3)$$
$$s = -1/3 \cdot u + 2/3 \cdot m \qquad\qquad (4)$$
$$p = 4/3 \cdot u - 2/3 \cdot m \qquad\qquad (5)$$

where u is the execution time on 1P/2GHz; m is the execution time on 4P/1GHz; s is the sequential component of execution time; and p is the parallel component of execution time.

The graph shows that programs are clustered into three categories. On the left are the highly-parallel programs ammp, apsi, mgrid, and wupwise. Notice that these are the same set of programs that were categorized into group one in Figure 1. Our analysis of sequential and parallel components showed that these programs have a large parallel component; hence, they benefit from running on a large number of low-power CPUs. On our four processor server, these benchmarks do not spend enough time in the sequential component to receive significant benefit from AMP. We surmise that on future large multiprocessors, the sequential component can be expected to become an increasingly dominant part of the total execution time (Amdahl's law), and therefore, the benefits of AMP can be expected to increase.

In the middle of are the moderately parallel programs art, BLAST, equake, FFTW, HMMER, and TPC-H. These benchmarks spend 23% to 36% of their execution time in sequential components. Ideally, all these benchmarks with significant amount of sequential component are expected to perform well on AMP. There are two reasons for the discrepancy between idealized and measured AMP performance. First, art and HMMER are not CPU bound (<70% CPU utilization) and hence, AMP provides less performance benefits than the potential shown in Figure 3. The second cause for the discrepancy is the thread migration overhead introduced by the AMP. The overhead may be amortized over long program phases such as in FFTW. Hence, benchmarks with long program phases show performance gains that are comparable to the idealized AMP performance.

The third category of benchmarks includes fma3d and swim. While these benchmarks have sequential components in the range of 31% to 54%, our visual analysis of these benchmarks revealed that these benchmarks have short sequential and parallel phases

and they switch rapidly between the two phases. Hence, the overhead introduced by thread migration in an AMP offsets any reduction in execution time. These benchmarks run well on a conventional 1P/2GHz processor.

6. Current Measurement and Simulation

In this section we present an alternative method for assessing the benefits of EPI throttling that employs a high-resolution measurement of CPU power consumption. While the AMP prototype described in the previous sections monitors CPU power with 1-bit of resolution (active or idle), this method offers 14 bits of resolution. The method comprises of two steps: (1) Supply current measurement and trace collection, and (2) Software throttle simulation. The software simulator models a hardware implementation of the EPI throttle that may use any of the EPI techniques presented in Table 1.

6.1   Supply Current Measurement

In the first step we measure the multiprocessor's supply current using a current probe and record the measured current over time. Our experimental setup consists of the following equipment: (1) 4-way Xeon 2GHz SMP server, (2) Agilent 34134A DC Coupled Current Probe, (3) Agilent 34401A Digital Multimeter, (4) Agilent 82357A USB/GPIB Interface, and (5) IBM ThinkPad T20 laptop.

Figure 4 shows our experimental setup. The SMP server is the same server used in the AMP prototype. In order to measure the supply current to the four processors the current probe is placed around the +12 volt wires between the power supply and motherboard. These wires feed the voltage regulators for the four Xeon processors. Note that the server board design constrained us to measure the input current to the voltage regulators rather than the output current that feeds the four CPUs. Hence, the measured current, converted to power, is the total for all four CPUs and includes small losses in the regulators.

The output from the current probe is sent to the Agilent 34401A multimeter. The multimeter captures 600 current measurement samples per second. We programmed the multimeter to collect samples for the same time as the run length of each of our 13 benchmarks. The Agilent 82357A USB/GPIB Interface is used to transfer the samples from the multimeter to the IBM ThinkPad T20 laptop. To measure the current flow over time we ran the unmodified benchmark binaries on the 4-way SMP. For each of the 13 benchmarks, between 20,000 and 400,000 samples were recorded.

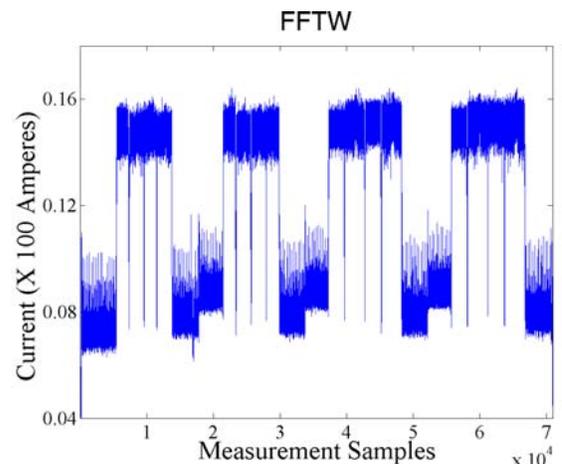**Figure 4: Current Measurement Setup**

Figure 5. Measured Supply Current on FFTW

An example of the measured supply current over time is shown in Figure 5 for FFTW. It is interesting to note the visual similarity between the current measurements in Figure 5 and the CPU utilization in Figure 2. The Y-axis represents current (in 1/100 amperes). The X-axis represents sample number. The phases of sequential (low current) and parallel (high current) execution are clearly apparent. Our measurements indicate that across all 13 benchmarks the four CPUs collectively consume 48 watts when idle and 220 watts when highly active.

6.2   Software Throttle Simulation

The software simulator reads the trace of measured supply current and simulates the actions of an EPI throttle that regulates all processors uniformly. The simulator computes the level of processor performance that is consistent with maintaining the total multiprocessor power within a fixed power budget. Intuitively, when power demands are too high the simulator creates the effect of slowing down all CPUs by reading the input trace more slowly. Reading the trace slowly makes everything run more slowly (CPU,

```
    int    timeclk   =0;       // simulator time
    double inputclk  =0;       // input trace time
    double intval    =1;       // integrator value
    double scalefactor=1;      // throttle amount
    double inpval;             // instantaneous power
    double diffval;            // difference between actual and desired power

    while (inputclk<datamax) {              // while there is data
      inpval=datatbl[(int) floor (inputclk)] *  // read current from trace
             scalefactor *                  // multiply by square or cube
             scalefactor *                  // of scalefactor to convert to
             (squareflag ? 1.0 : scalefactor);  // power

      diffval = inpval - threshold;         // subtract power threshold
      intval += diffval*gain;               // multiply by gain and accumulate

      if (intval<lowerclamp) intval=lowerclamp; // saturate at lower clamp (usually 1.0)
      if (intval>upperclamp) intval=upperclamp; // saturate at upper clamp

      scalefactor = 1.0/intval;             // compute new scalefactor

      inputclk += scalefactor;              // advance trace time
      ++timeclk;                            // advance simulator time
    };
```

Figure 6. EPI Throttle Simulator Algorithm

memory, I/O subsystem). By choosing the 13 benchmarks that are mostly CPU bound in this paper we minimized the adverse effects of slowing components in the system other than the CPU. The simulator outputs an execution time measured in sample intervals. The wupwise benchmark for instance when run on all four Xeon processors at 2GHz consumes 220 watts and generates about 400,000 current measurement samples (equivalent to 670 seconds of execution time). However, if the power was constrained to 55 watts the same trace would have executed in 744,489 sample intervals, equivalent to 1,240 seconds of execution time.

The details of the simulator algorithm are shown in Figure 6. The algorithm takes three inputs: the current measurement trace, the power threshold value, and a gain constant for the feedback loop in the throttle. The current trace is stored in the datatbl array. Inputclk is the time (measured in sample intervals) in the input trace; timeclk is the time (measured in sample intervals) of the simulator output. As mentioned earlier our equipment collects 600 samples per second and hence each sample interval corresponds to 1/600th of a second. The simulator is capable of scaling the measured current by either the square or the cube of the throttle amount. For our work, we use the square relationship since we are conservatively assuming that power can be made proportional to the square of the performance. The second input is the threshold constant that sets the desired power budget. The threshold value is usually selected by the designer to be the maximum power consumed by the chip. In our experiments we used 55 watts as the power threshold that can be dynamically distributed amongst the four processors. Our measurements showed that the maximum power consumed on our four processors running at 2GHz is 220 watts. Since we set the power budget to be the same as one processor running at 2GHz we used 55 watts (¼ of the 220 watts) as the power threshold. The third input to the algorithm is the gain constant, which determines how quickly the throttle's feedback loop can respond to changes in the power consumption. We chose a value of 0.1 which enables the processor's throttled performance to double or halve in roughly 1/10 of a second. Finally, to prevent the throttle from attempting to run the CPUs faster than possible during periods of low power consumption, the lower clamp is set to the baseline value of 1.0, corresponding to our 2GHz processor. An optional upper clamp is also provided but was never triggered in our simulation runs.

In Figure 6 inputclk is advanced each iteration by the scale factor. The scale factor is never greater than one, since intval is clamped at a maximum of one. Typically the scale factor is less than one and hence, the same input current data may be read several times during consecutive iterations. Thus, we simulate the effect of reducing the processor's power and correspondingly its performance.

6.3 Software Throttle Simulation Results

The results produced by the EPI throttle simulator are shown in Figure 7. This figure shows the normalized performance on each benchmark as the power threshold

is varied from 0 to 220 watts. A reference point is plotted illustrating our rule-of-thumb relationship between performance and power (½ relative performance at ¼ relative power, which is 55 watts). Several interesting observations can be made from this figure. The graphs show that the simulated performance for all the benchmarks is greater than this reference point; the EPI throttle takes advantage of the fact that the four processors are not consuming maximum power at all times. The set of programs art, BLAST, FFTW, HMMER and TPC-H show the least performance degradation with reduced CPU power indicating that these benchmarks have phases of execution where the four CPUs are underutilized. Hence, these benchmarks are expected to perform well on our AMP prototype, which concurs with the results in Figure 1. Finally, as expected, the CPU-intensive programs such as wupwise and applu show the greatest performance degradation as CPU power is constrained.
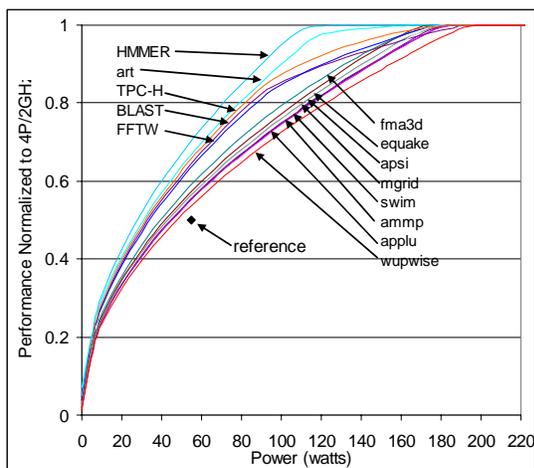

Figure 7: Performance versus Power Threshold

6.4   Prototype versus Simulation Results

The AMP prototype uses thread-level parallelism within a program to determine the number of processors and the corresponding frequency (duty cycle) to use to stay within the power budget. However, we expect in practical implementations an EPI throttle would monitor the activity levels of the various blocks within a chip, and based on activity determine when to reduce the power consumption. The advantage of the AMP prototype is that one can quickly measure the performance impact of a throttle using the wall clock execution times. The throttle simulator, on the other hand, is a more flexible implementation because of programmable parameters.

Figure 8 compares the measured performance using the AMP prototype with the simulated performance using the EPI throttle simulator. Across the suite of benchmarks, the two approaches provide comparable results. However, a small number of benchmarks have noticeable differences. These differences may be due to the following effects:

1. Programs with frequent transitions between sequential and parallel phases (such as fma3d) may run slowly on the AMP due to the added overhead of thread migration
2. Programs with low activity levels in all four CPUs during the execution of parallel regions will run faster on the simulator because the EPI throttle doesn't need to slow down the CPUs by much to reduce power.
3. Programs with significant accesses to main memory and I/O may run slower on the simulator because the simulator uniformly slows down all components of the computer, including processors, memory, and I/O.

The close correlation of the results in Figure 8 between the measurement-driven simulator and the physical AMP prototype system shows that a realistic EPI throttle can deliver the performance gains in practice.
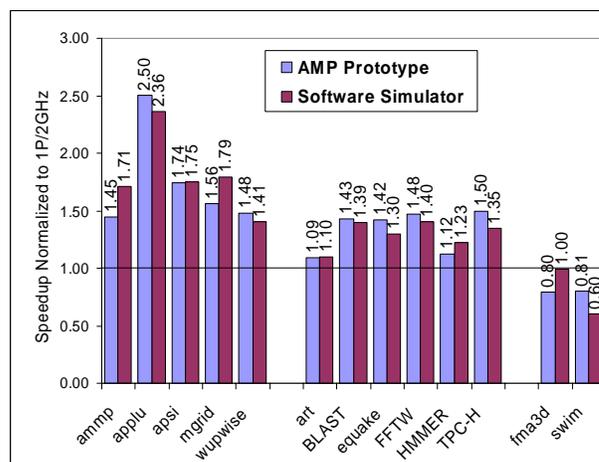

Figure 8: Prototype and Measurement Results

7. Conclusion and Future Work

In this paper we have presented a comprehensive performance evaluation of EPI throttling using a physical AMP prototype system and current measurements on a physical system. We use clock throttling and processor affinity to prototype an asymmetric multiprocessor (AMP) using an off-the-shelf 4-way SMP. Each processor in the AMP expends varying amount of EPI based on the available thread-

level parallelism. By running a wide range of multi-threaded programs on the AMP prototype, we show a 38% wall clock speedup compared to a 4-way SMP with a fixed power budget. The performance advantage is due to the EPI throttle's ability to dynamically allocate a fixed power budget among a small number of high-performance (high EPI) processors and a large number of low-power (low EPI) processors in order to rapidly execute both the sequential and parallel portions of the computation.

In order to improve the 1-bit resolution of EPI throttling in AMP, we measure the supply current on a physical 4-way SMP server running multi-threaded programs. We feed the measured data as input to a software simulator that implements a more flexible EPI throttle. We show that speedup from the AMP prototype is comparable to the speedup achieved using the flexible EPI throttle.

We believe that future chip multiprocessors will incorporate some form of EPI throttle in order to deliver maximum performance while keeping power consumption within a fixed budget. By measuring wall clock speedups, we show that an EPI throttle works well. Furthermore, using two complementary methods we make a compelling case that an EPI-throttled chip multiprocessor is effective in mitigating the effects of Amdahl's law when running multi-threaded programs with non-trivial sequential components.

We now discuss areas for future work. While our work is based on a 4-way multiprocessor, future multiprocessors may be expected to contain very large numbers of processors. *How well can the EPI throttle be expected to work with future systems and software?* As chip multiprocessors with larger and larger numbers of processors become practical, we expect the potential performance benefits of EPI throttling will increase. This is due to Amdahl's law – as the parallel phase is divided among more and more CPUs, it becomes increasingly important to run the sequential phase quickly. Since the effectiveness of the EPI throttle is highly dependent on the characteristics of software, we may ask *what percentage of a typical software workload consists of an inherently sequential component?*

Hardware implementation is another area for investigation. *What is the best microarchitecture for the EPI throttle?* The EPI throttle consists of a mechanism to monitor the multiprocessor's activity, a feedback loop, and a mechanism to control the multiprocessor's EPI as described in Table 1. We expect the EPI throttle to be implemented as a hardware mechanism that operates transparently to software. Software sees a symmetric multiprocessor with an unusual property: individual threads become slower as software asks hardware to run more threads, even though net throughput increases. The EPI throttle can make software execution times hard to predict and raises possible fairness issues. *What are the software implications of the EPI throttle?*

In our prototype, we studied the effects of EPI throttling on CPU performance and power. A more comprehensive study may choose to consider the entire platform, including the main memory and I/O subsystem. *How does an EPI throttle account for platform-level power interactions?*

Finally, in our work we've taken the simple goal of keeping the total CMP power constant, within a fixed power budget. Future EPI throttles may need to operate under several competing constraints. These may include minimizing energy, minimizing di/dt-induced supply voltage variation, reducing the magnitude of thermal hot-spots, or guaranteeing a certain quality of service. *What are the most appropriate goals for future deep submicron processors, and how should an EPI throttle function given multiple, potentially conflicting goals?*

We believe that the advent of large chip multiprocessors, operating in a power-constrained environment, with performance characteristics governed by Amdahl's law, has opened up an exciting new area for future research.

8. Acknowledgments

9. References

[1] D.H. Albonesi, R. Balasubramonian S.G. Dropsho, S. Dwarkadas, E.G. Friedman, M.C. Huang, V. Kursun, G. Magklis, M.L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P.W. Cook and S.E. Schuster. Dynamically tuning processor resources with adaptive processing. In *IEEE Computer*, 36(12):49-58, December 2003.

[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers and D.J. Lipman. Basic local alignment search tool. In *Journal of Molecular Biology,* vol. 215, pages 403-410, 1990.

[3] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W.B. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer

Performance. In *Proceedings of the Workshop on OpenMP Applications and Tools*, Lecture Notes in Computer Science, vol. 2104, pages 1-10, July 2001.

[4] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 282-293, June 2000.

[5] D. Brooks and M. Martonosi Dynamic thermal management for high-performance microprocessors. In *proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 171-182, January 2001.

[6] T.D. Burd and R.W. Brodersen. Energy Efficient CMOS Microprocessor Design. In *Proceedings of the 28th Annual Hawai'i International Conference on System Sciences*, vol. 1, pages. 288-297, Jan. 1995.

[7] FFTW: http://www.fftw.org

[8] R.J.O. Figueiredo and J.A.B. Fortes. Impact of heterogeneity on DSM performance. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture*, pages 26-38, January 2000.

[9] E. Grochowski, R. Ronen, J. Shen, H. Wang. Best of Both Latency and Throughput. In *Proceedings of the 22nd International Conference on Computer Design,* pages 236-243, October 2004.

[10] S.H. Gunther, F. Binns, D.M. Carmean, J.C. Hall. Managing the Impact of Increasing Microprocessor Power Consumption. *Intel Technology Journal, First Quarter* 2001. http://www.intel.com/technology/itj/q12001.htm

[11] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Willey, M. Chen, M. Kozyrczak, and K. Olukotun. The Stanford Hydra CMP. *Hot Chips 11*, August 1999.

[12] HMMER: http://hmmer.wustl.edu

[13] Intel® Pentium® 4 Processor in the 423-pin Package at 1.30 GHz, 1.40 GHz, 1.50 GHz, 1.60 GHz, 1.70 GHz and 1.80 GHz Datasheet,. http://support.intel.com/design/pentium4/datashts/249198.htm, pages 78-79, 2001.

[14] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proceedings of the 36th International Symposium on Microarchitecture,* pages 93-104, December 2003.

[15] J. Kahle. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum '99*, October 1999.

[16] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan and D.M. Tullsen. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In *proceedings of the 36th International Symposium on Microarchitecture*, pages 81-92, December 2003.

[17] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi and K. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings 31st International Symposium on Computer Architecture*, pages 64-75, June 2004.

[18] J. Li and J.F. Martínez. Power-Performance Implications of Thread-level Parallelism on Chip Multiprocessors. To appear in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, March. 2005.

[19] S. Manne, A. Klauser and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings the 25th International Symposium on Computer Architecture*, pages 132-141, June 1998.

[20] D. Menasce, V. Almeida. Cost-performance analysis of heterogeneity in supercomputer architectures. In *Proceedings of Supercomputing*, pages 169-177, November 1990.

[21] T.Y Morad, U. Weiser and A. Kolodny. ACCMP - Asymmetric Chip Multi-Processing. CCIT Technical Report #488, http://www.ee.technion.ac.il/morad/publications/accmptr.pdf, June 2004

[22] M. Stonebraker and L.A. Rowe. The design of POSTGRES. In *Proceedings of the International Conference on Management of Data*, pages 340-355. June 1986.

[23] TPC-H: http://www.tpc.org/tpch

[24] J. Tschanz, S. Narendra, Y. Yibin, B. Bloechel, S. Borkar, D, Vivek. Dynamic-sleep transistor and body bias for active leakage power control of microprocessors. In *IEEE Journal of Solid-State Circuits*, 38(11):1838-1845, November 2003.