Dynamically Configurable Caches in Low Power Computing

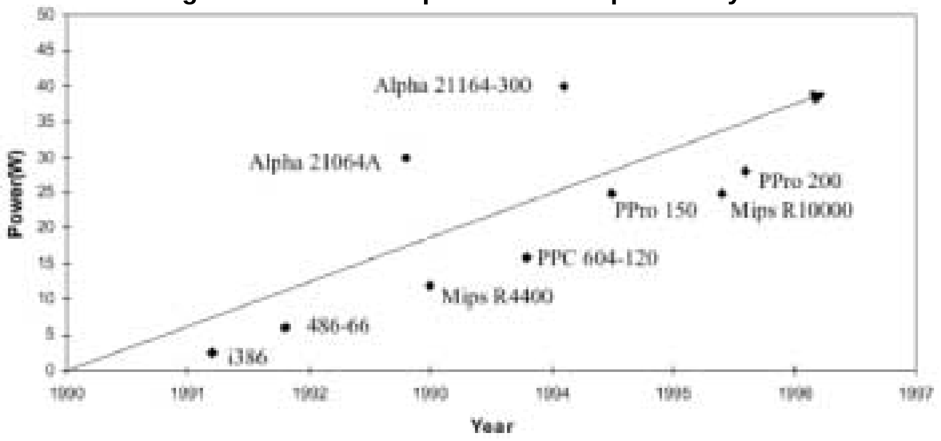
Authors:	Michael C. Brogioli
	Bryan Jones
Professor:	Scott Rixner
Date:	April 24, 2001
Course:	ELEC/COMP 525

Table of Contents

1.0 Motivation

As we rapidly approach the milestone of one billion transistors on a single chip, the subject of power consumption is becoming an increasingly important issue. No longer is it possible to cool a modern microprocessor with simple heat sinks and fans, and with the recent explosion in mobile computing the problem is only exacerbated. Figure 1 below shows power consumption trends of general purpose processors from 1990 - 1997.[1] Extrapolating this data out to the year 2010 shows that, even by conservative estimates, general purpose processors will be consuming well over 100 Watts of power. This level of power consumption is unacceptable in portable devices for two reasons. First, barring any unforeseen advances in battery technology, the idea of all day computing becomes unachievable due to the amount of power required by the system. Secondly, due to the power consumption per unit area of die, the means of cooling the processor begins to seriously hinder the portability of such a device.

Figure 1: Processor power consumption vs. year



In recent years, architectures have emerged which claim to drastically reduce processor power consumption. In the authors' opinion, however, many of these techniques appear to be rather brute force in nature and improperly focused. One popular trend is to allow the processor to run at multiple frequencies, switching to the lower clock speeds when attempting to conserve power. Examples of this include Intel's SpeedStep technology in the Pentium III processor, and Transmeta's LongRun technology in the TM5400 series processors.[2,3] Although scaling clock frequency clearly reduces overall processor power consumption, in many cases system performance suffers accordingly. In this paper, we propose a solution which attacks the issue with a finer level of granularity, i.e. at the architectural level, and is cognizant of system performance. By doing this, we hope to decouple processor power consumption and application performance.

2.0 Hypothesis

Looking at the transistor budgets for today's modern microprocessors, excluding vector based architectures, it is clear that the majority of transistors are being used in caches of various types[4,5]. It is no surprise then, that a major portion of processor power is consumed by the cache due to repeated cycling of the clock to cache cells, regardless of whether they hold any useful information or not. Furthermore, with today's applications ranging from multimedia and word processing applications to scientific workloads it, should be no surprise that caching needs are not homogeneous across the board.

In this paper, we propose a data cache which can be dynamically reconfigured at runtime with respect to an application's memory footprint. By dynamically reallocating data cache resources, areas of the data cache which have not been allocated can be "turned off" to conserve power. The availability of such an architectural feature makes the following property important: given an arbitrarily large data cache, an application will achieve a given level of performance. Further increasing the data cache size will no longer improve application performance, as the entire working set of the application now resides in data cache and storage space is no longer an issue.

Combining the information given above, we hypothesize that there exist cases in which the amount of memory referenced by an application over a given period of time is far smaller than the total data cache size. By profiling an application's memory access patterns over various input data sets, we hope to gain insight into not only it's overall data cache requirements, but also the data cache requirements at various points throughout it's execution. Knowing the data caching requirements throughout the application, we can dynamically reallocate subsets of the total data cache such that the majority of the application's working set fits in data cache. If the working sets of the application are smaller in size that the total available data cache, then we should be able to reduce power consumption considerably due to the amount of non allocated data cache, which will be powered down during runtime. Furthermore, since the lower bound on the allocated data cache size is approximately the active working set of the application, performance loss should be minimal.

We anticipate the greatest power savings to be on multimedia type applications due to the small amount of temporal locality for cached data. Scientific applications will most likely yield minimal results. This is expected due to the fact that many of these type applications use very large data sets, and are compiled with explicit knowledge of the underlying processors cache resources. Techniques such as loop blocking are often employed when a loops working data set is larger in size than the processors data cache, and very often make efficient use of the available data cache.

3.0 Architecture

All simulations were performed with the Simplescalar simulator, version 3.0, available for download at www.simplescalar.org. In order track the amount of power consumed by each of the processors resources, the Wattch tool set version 1.02 was integrated into Simplescalar as well. The Wattch toolset is available for download at www.ee.princeton.edu/~dbrooks/.

Table 1 below lists the parameters used during Simplescalar simulations. Parameters were chosen to closely resemble the features of many modern general purpose processors.

Table 1: Simplescalar Simulation Parameters

Simulation Parameters	Values
L1 inst. cache size	Varied, 8K – 128K
L1 inst. cache structure	Varied, direct mapped, fully associative, two way, four way
L1 inst. cache replacement policy	Least recently used
L1 data cache size	Varied, 8K – 128K
L1 data cache structure	Varied, direct mapped, fully associative, two way, four way
L1 data cache replacement policy	Least recently used
L2 unified cache size	Fixed, 256K
L2 unified cache structure	Four way associative
L2 cache sets	64
L2 cache line length	1024 bytes
Process technology	0.35 μm
Clock speed	600 MHz
Clocking method	Aggressive conditional, with leakage.
Integer ALU's	4 + 1 integer multiply
Floating point ALU's	4 + 1 floating point multiply
Memory ALU's	2
Decode / issue / commit width	4 instructions
Register update unit	16 entries
Load / store queue size	8 entries
Branch predictor	bimodal, 2K entries + 8 entry return buffer, 512 entry BTB
TLB size	4K
TLB structure	Four way associative
TLB replacement policy	Least Recently Used

In order to facilitate the powering down of the data cache, we propose the use of multiple address decoders interfaced to the data cache via a single multiplexor. If the data cache can be run in one of four different sizes, for example, then we propose the use of four decoders. Each decoder will map values into the data cache according to the active data cache size. This implementation will work well when the active data cache size is scaled down, but when the active data cache size is increased, complications may arise. Consider the following: when the data cache size is increased, a new address decoder is selected to map addresses into data cache. Values that resided in data cache before the data cache's size was increased will still reside in the same locations. The new decoder, however, may map the same addresses to locations recently turned on and thus register a data miss due to the fact that the same data was loaded into the cache earlier, but by a different decoder. This issue is considered beyond the scope of our research, however the topic is further addressed in the conclusions section.

The powering down of the data cache itself consists of simply stopping the cycling of the clock to areas of the data cache which are not to remain active. Data held within this portion of the cache will be preserved, and again be accessible if and when this portion of the data cache is again made active. Some leakage of clock current is expected, however, most likely on the order of five to ten percent. This is addressed with the clocking method parameter in table 1 above.

Because the amount of transistors added to the baseline architecture is minimal, neither die area nor increased power consumption should not be much of an issue . Though multiple address decoders will be required for the data cache, the transistor budgets of modern architectures will more than accommodate this. The use of a multiplexor interface between the address decoders and the data cache prevents the need for additional read and write ports on the data cache. By not adding ports to the baseline data cache, cache hit time will be preserved assuming the additional delay of the multiplexor and address decoders is negligible. Due to the simplicity of this design and the fact that the underlying data cache is not modified, this implementation should scale well to future architectures.

4.0 Experimental Framework

The project was divided into multiple stages of experimentation, each of which built upon the results obtained in the previous stage. By working in this manner, it was unlikely that research would pursue down a dead end path, and be based on improper speculation as opposed to known fact. If the previous stage of experiments didn’t suggest what had been anticipated, perhaps the direction of research should be reconsidered. Furthermore, by breaking the project down into more manageable milestones, it was less likely that research would fall behind scheduled deadlines.

All experiments were performed on a subset of benchmarks taken from the SPEC 2000 suite. This working subset consisted of 188.ammp, 181.mcf, 197.parser, and 175.vpr. Only four benchmarks were used due to the number of simulations which were to be run, the amount of computation time each simulation required, and the time frame of the project. A brief description of each benchmark is given in table 2 below.

Table 2: Benchmark types and descriptions

Benchmark	Type	Description
188.ammp	Floating point	Molecular dynamics simulation.
181.mcf	Integer	Vehicle scheduling in public mass transportation system.
197.parser	Integer	Syntactic parser of English language.
175.vpr	Integer	Integrated circuit CAD design program.

5.0 Validity of Hypothesis

The first round of experiments were designed to test whether or not the proposed hypothesis was valid. More specifically, the issues of what percentage of total processor power was dedicated to powering the data cache, and how much of the processors data cache were these benchmarks actually using. To determine what percentage of total processor power was being consumed by the data cache, we ran simulations of each of the benchmarks with fully associative L1 data and instruction caches of size 8KB, 16KB, 32KB, 64KB, and 128KB each, keeping all other parameters fixed. Figure 2 below show how much power each of the processors major resources consumed as data cache size varied for the 181.mcf benchmark. Table 3 contains the symbol key for figure 2. The results for the 181.mcf benchmark were typical of each benchmark used, thus we have chosen to omit plots for each benchmark individually.

Figure 2: Processor resource power consumption for 181.mcf

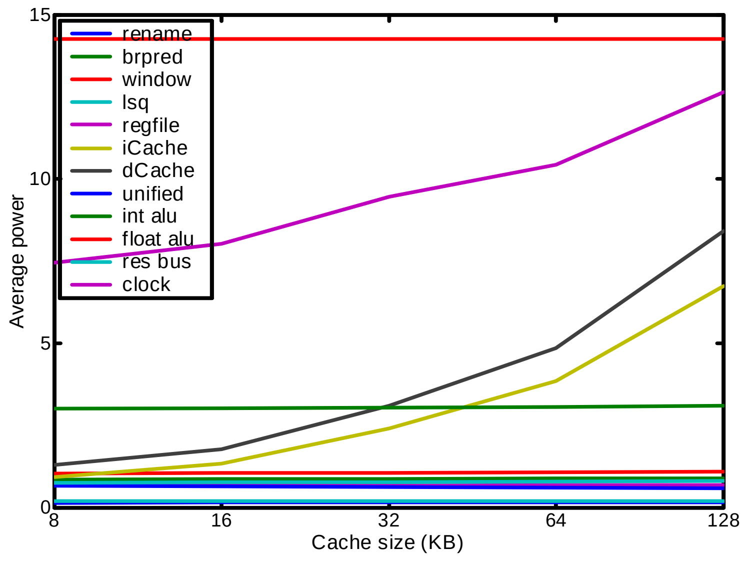


Table 3: Key of Names for Figure 3

Name	**Processor Resource**
Rename	Rename buffer
Brpred	Branch prediction hardware
Window	Instruction fetch, decode window.
Lsq	Load store queue
Regfile	Register file
Icache	L1 instruction cache
DCache	L1 data cache
Unified	Unified L2 instruction and data cache
Int Alu	Integer alu's
Float Alu	Floating point alu's
Res Bus	Result bus
Clock	Processor Clock

In figure 2 we can see that when L1 data cache is less than 16KB, its power consumption is substantial but not overwhelming. When L1 data cache is increased to the 64KB - 128KB range however, it becomes the dominating factor in overall processor power consumption. This proves that the cache is one of the more power hungry processor resources today, and if we are to begin reducing overall processor power consumption this is prime area in which to focus our efforts.

In order to estimate the amount of total L1 data cache being utilized by these benchmarks, we ran simulations of each of the benchmarks with fully associative, two way, and four way associative L1 data and instruction caches. In addition to varying the cache structure, we again varied the L1 data and instruction cache size using values of 8KB, 16KB, 32KB, 64KB, and 128KB. For each of the simulations we measured average IPC and average power consumption. Figures 3a - 3d below show the results obtained for average IPC, and figures 4a –4d show the results obtained for average power consumed.

Figure 3a: Average IPC for 181.mcf

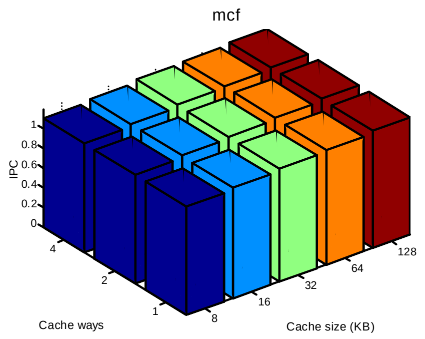


Figure 3b: Average IPC for 175.vpr

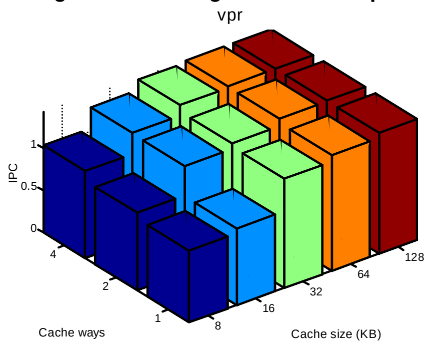


Figure 3c: Average IPC for 188.ammp

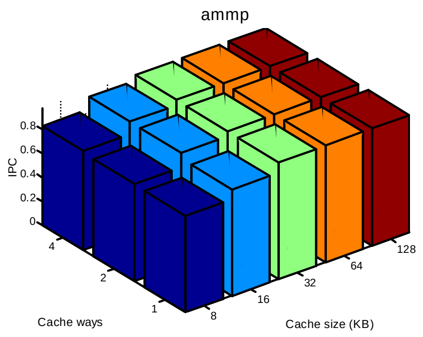


Figure 3d: Average IPC for 197.parser

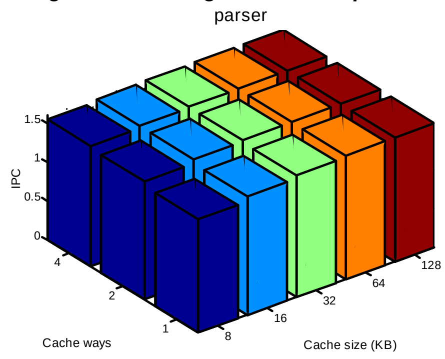


Figure 4a: Average Power for 181.mcf

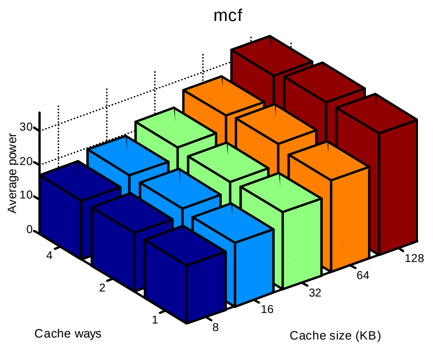


Figure 4b: Average Power for 175.vpr

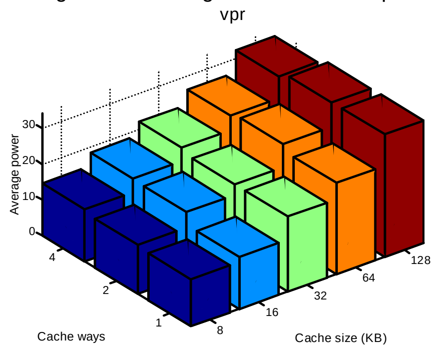


Figure 4c: Average Power for 188.ammp

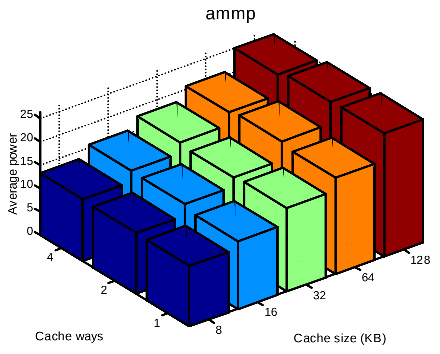
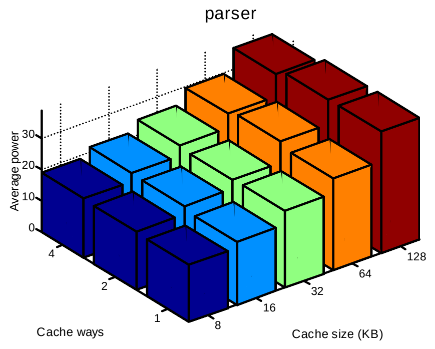


Figure 4d: Average Power for 197.parser



For 175.vpr, it is apparent that anything larger than a 16KB L1 data and instruction cache is not necessary, as there is no improvement to average IPC above this threshold. Looking at the corresponding average power consumption for 175.vpr, by operating with 16KB L1 data and instruction caches will reduce total processor power consumption by almost fifty percent, with minimal drop in IPC. Going so far as to further reduce L1 data and instruction cache sizes to the 8K range reduces IPC somewhat, but reduces total processor power consumption by as much as sixty six percent. In the other benchmarks the decrease in IPC was less drastic than in 175.vpr, while reduction in power consumption was similar. From this we conclude that even better results are attainable, and 175.vpr exhibits the most conservative of the four cases.

6.0 Experimental Analysis

Expanding upon the idea that there exist cases in which application performance can be maintained while data cache size is decreased, it became apparent that a more in depth analysis of an applications caching behavior was necessary. At this point it was decided that further research efforts would focus only on fully associative data caches. This decision was made with regard to the large amount of programming and debugging that was necessary, as well as the increasing number of simulations that would be required, each of which costing vast amounts of computation time. Additional comments on multiple way and direct mapped data caches will be further addressed in the conclusions and future work sections of the paper.

The first step in obtaining a better understanding of caching behavior was to obtain execution traces for the suite of benchmarks used. Modifications were made to Simplescalar's sim-outorder simulator to generate a trace of load and store operations for a given benchmark's execution. Care was taken to only consider those operations which were committed rather than speculatively executed and discarded. In addition to creating a trace of memory operations, a simulator for arbitrarily sized fully associative data caches was incorporated into sim-outorder as well, hereupon referred to as the ideal data cache. Ideal data cache line length was chosen to be of unit size, where a unit can be any arbitrarily sized data type, for reasons that will soon become apparent.

The replacement policy for the ideal data cache is derived from L.A. Belady's offline page replacement policy, better known as the MIN algorithm[6]. This algorithm was chosen due to the similarities between mapping memory loads into a fully associative cache with unit sized line length and register allocation over straight line blocks of code. If one considers the benchmark's memory operation trace to be a straight line block of code, and each unit sized cache line to be a register, then the problem becomes one of register allocation over basic blocks. The algorithm works as follows: the ideal data cache is initialized to be empty. At each load instruction, data is loaded into an empty ideal data cache line. Once all ideal data cache lines are full, at each subsequent load operation a data value currently in ideal data cache must be evicted. In order to determine which value to evict from the ideal data cache, the algorithm checks to see if any ideal data cache entries are never referenced again further ahead in the instruction trace. If there exist entries which are never referenced again, then one of these entries is evicted and the data from the outstanding load instruction is stored in said cache line. If all entries currently in the ideal data cache are referenced further ahead in the instruction trace, then the cache line holding the data which is referenced furthest ahead in the instruction trace is evicted, and the data from the outstanding load instruction is stored in said cache line.

Simply knowing what data resides in the ideal data cache at any given time does not provide adequate insight into an application's caching behavior, however. Combining the knowledge of what data resides in the ideal data cache at any given time with the knowledge of what percentage of the data residing in the ideal data cache is actually referenced again in the upcoming instruction stream is far more beneficial. In order to gather this information, additional modifications were made to the sim-outorder simulator. A feature named "liveness" was added which analyzes the contents of the ideal data cache and reports the number of ideal data cache entries which are referenced above a specified threshold number of times in the upcoming specified number of instruction. This tool also works off of the generated memory reference instruction traces mentioned earlier. Figures 5a - 5d below show the results of the liveness analysis for each benchmark in the suite. It should be noted that due to the amount of computation time it required to run simulations with liveness information, the plots for all but 175.vpr are incomplete. They have been included to illustrate caching trends however. Liveness analysis was run using a 64KB fully associative ideal data cache. Cache contents were analyzed after every five thousand instructions, and the instruction look ahead window was twenty thousand instructions. A value residing in ideal cache was counted as useful if it was referenced once or more in the look ahead window.

Figure 5a: Liveness plot for 181.mcf

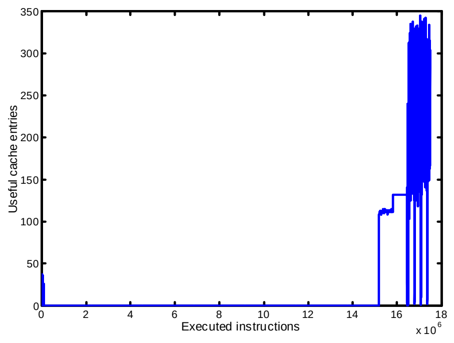


Figure 5b: Liveness plot for 175.vpr

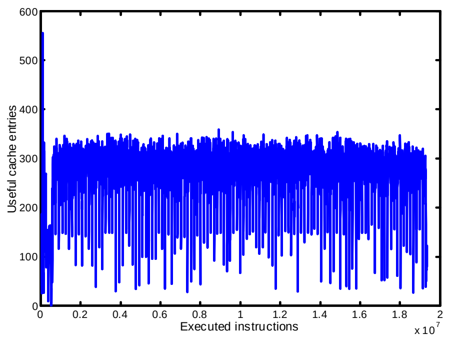


Figure 5c: Liveness plot for 188.ammp

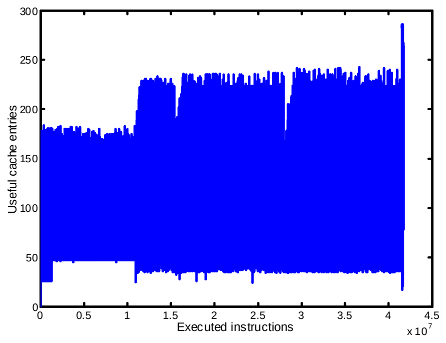
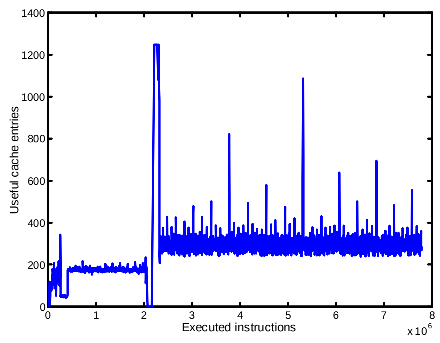


Figure 5d: Liveness plot for 197.parser



Clearly, in 181.mcf there is very little temporal locality in cached data until the end of the plot, where the spikes occur. For this case, very little if any data cache is needed during the beginning of computation. Looking at 188.ammp, there are just over 200 useful values in cache until approximately ten million instructions have been executed, at which point that figure jump to just below three hundred. These results correspond with our initial hypothesis that there are significantly sized portions of execution time in which the amount of useful data stored in the L1 data cache varies. It is unfortunate, however, that our simulations required so much time and more data could not be analyzed. In hind sight, this is due to the overall method chosen to analyze cache data. Simply analyzing the data at fixed intervals of instructions requires far to much computation time, as time is directly proportional to simulated cache size and look ahead window size. A better implementation would have been to profile the benchmarks, as opposed to creating traces for them. Data cache requirements could then be determined on a procedure by procedure basis. Considering only those procedures which execute for significant portions of time, we could then tailor the data cache size accordingly.

Using the information in figures 5a – 5d above, we ran simulations for all of the benchmarks while varying data cache size throughout program execution. The framework for measuring power consumption while dynamically enabling and disabling processor resources worked as follows. At the start of a simulation, the Wattch toolset initializes data cache power consumption coefficients according to the size and configuration of the data cache, and initializes data cache power consumption to zero. At each data cache access, the simulated data cache access cost is added to the running total, based on the calculated power consumption coefficients. In order to model dynamic data cache access costs, whenever data cache size was changed, the coefficients used to determine data cache access cost were recalculated. The new cost of a data cache access was then added to the running total cost at each access.

In addition to accounting for the varying power costs for a data cache access, it was also necessary to simulate what happened to data residing in data cache when the data cache was resized. A pessimistic model was chosen in which during a data cache size switch, all data residing in cache was flushed. Unfortunately, due to an error which was caught in our earlier simulations, time which had been allocated to simulating the dynamically resizable data cache had to be used to correct for the earlier simulations.

We hypothesized that by dynamically resizing the L1 data cache at various point during program execution, we would be able to maintain average IPC while reducing the total power consumed by the processor. It was already proven that IPC could be maintained while data cache size was reduced from 128K, even if data cache was not dynamically resized during program execution. Looking at the plots of liveness information in figures 5a –5d, if a line were drawn horizontally across the plot at the level of the highest peak, one could conceivably consider all area between the horizontal line and the peaks in the plot to be error, or wasted power. We originally hoped to be able to remove this wasted power by resizing the data cache dynamically.

Although results for dynamically resizing the L1 data cache were not obtained, there are a number of things which should be considered. The first of these is the liveness data obtained for each benchmark. Assuming that liveness data was available for the entire execution of the benchmark, it may not have proved as useful as was originally hoped. One reason is that even if architectures L1 data cache were fully associative, if power consumption were an issue it would most likely not be designed with single unit line length. This is due to the fact that if line length were of unit size, there would be as many tags to check on each access as there were lines in the data cache. The power cost of repeated tag checks would be quite large. Additionally, if non unit line length were used, the issue of spatial locality would come into play. If there were large amounts of spatial locality in the cache data, then multiple values reported by liveness would most likely map to the same cache line. If there were little spatial locality, however, it is likely that even if the fully associative data cache were somewhat larger than what liveness reported, not all data values reported by liveness would fit into data cache.

Another point to consider is the reduction in processor power consumption achieved by dynamically resizing the L1 data cache. Early on, the total power consumption of the processor as a function of cache size and structure was shown. Although the reduction in power consumption was substantial, one must keep in mind that both L1 data and instruction caches were resized. It is true that L1 data cache consumed more power than L1 instruction cache on average, but if the simulations only varied L1 data cache size while L1 instruction cache size was held static, it would be very difficult if not impossible to achieve such drastic results. Another factor attributable to varying the L1 instruction cache size in parallel with L1 data cache size is the reduction in average IPC seen in figures 4a – 4d. Because L1 instruction cache size was varied, instruction cache misses which occurred due to the smaller L1 cache sizes may have exaggerated the decrease in average IPC seen. If L1 instruction cache size were held constant at 128KB, a higher average IPC would most likely have been seen.

8.0 Conclusions

Although our final experiments did not complete in time, it was proved early on that our hypothesis was indeed correct. For many benchmarks, data cache size can be reduced while maintaining average IPC and reducing total processor power consumption. Were additional time available, however, there are a number of refinements which could be made to the research presented. The first of these would be to run simulations again to determine average IPC and power consumption while varying L1 data cache size and holding L1 instruction cache size fixed. These results would provide clearer insight into the total results achievable. In addition to this, restructuring of the liveness analysis software would provide faster simulation times, The option to run liveness analysis over varied types of caches would prove to be useful as well. This would eliminate the speculation involved in determining how direct mapped cache performance relates to two and four way cache performance. A less pessimistic model of data cache size switching in the simulator, one which more closely correlates to our proposed architecture, would incur less penalty for data cache size switching as well. Finally, investigation into dynamically resizing data caches which are not fully associative would provide useful insight into this technology's applications in modern architectures.

The results presented in this paper may not be as drastic as originally hoped, but that should not overshadow the fact that it is indeed possible to reduce processor power consumption via configurable caches. Even if data cache size is not dynamically reconfigured during program execution, statically setting the size before an application runs will provide significant improvements. Furthermore, with the advent of billion transistor chips, cache size is only going to expand. As this happens, increasing amounts of power will be consumed by the cache, and the benefits of the proposed technology will grow. With some additional research, there is no reason that these techniques can not be applied to instruction caches as well. In conclusion, we have only touched upon the benefits achievable with the use of dynamically configurable caches. We believe that architectural and compiler based advancements in the upcoming years should make such a technology feasible in many general purpose processors targeted at the low power market.

Table of Figures

Table of Sources

1.) Professor Scott Rixner, Elec/Comp 525, Spring 2001, Rice University, “Lecture 1: Technology Overview”, page 20.

2.) Intel Corporation, “Intel Delivers Breakthrough in Microprocessor Power Consumption”. www.intel.com/pressroom/archive/releases/20010130comp.htm.

3.) Tom R. Halfhill, “Transmeta Breaks x86 Low-Power Barrier,” Microprocessor Report (Februry 14, 2000), pages 4-5.

4.) Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, Jared Stark, “One Billion Transistors, One Uniprocessor, One Chip,” IEEE Computer, September 1997, page 52.

5.) Keith Diefendorff, “Power4 Focuses on Memory Bandwidth”, Microprocessor Report (October 6, 1999), page 3.

6.) L. A. Belady, “A study of replaceent algorithms for a virtual storage computer,” IBM Systems Journal, pages 78-101, 1966.